EXPLORATION OF COMMUNICATION INTERCONNECTION NETWORK CONGESTION AND METHODS OF MITIGATION THROUGH SIMULATION

Neil McGlohon

Submitted in Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

Approved by:
Mark P. Shephard, Chair
Christopher D. Carothers, Research Advisor
George M. Slota
Robert B. Ross

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[August 2021]
Submitted June 2021

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

While it is my name on this document, it has required the help and support of many people, to whom I am very grateful and would like to thank:

This work was completed with the aid, in no small part, of my advisor, Christopher Carothers. As a font of knowledge in this field, he has played a vital role as a guide, consultant, and advocate every step of the way.

Academic collaborators and co-authors that contributed to the work included in this document are Misbah Mubarak, Kevin Brown, Robert Ross, Noah Wolfe, Sudheer Chunduri, Scott Hemmert, Michael Levenhagen, Eric Borch, Ram Huggahalli, Scott Parker, Malek Musleh, Kalyan Kumaran.

On countless occasions have I also sought the academic, professional, and personal advice of my colleagues at RPI: Ben Horne, Caitlin Ross, Mark Plagge, Erika Mackin, Brayden Hollis, Jeramey Tyler, and Matt Peveler. I've leaned on my close friends Zach Higgins and Alicia Bernson time and again; they're a primary reason why I have considered Troy my home.

In the years prior to coming to RPI, I was given the opportunity to serve as an undergraduate research assistant in the Department of Physics at the University of Oklahoma. My research advisor at the time, Sheena Murphy, was an incredibly influential figure and gave me my first real glimpse into the world of academic research. Another significant mentor during my time at OU was my advanced laboratory professor and senior capstone advisor, Matthew Johnson, who constantly challenged me to ask better questions to find better answers. The lessons learned and foundations developed during those years have been absolutely critical in any academic success thereafter.

Much of my interest in science and exploration can be directly attributed to numerous role models, teachers, and educators I had as a child and through young adulthood in school and at Science Museum Oklahoma/Omniplex. Specifically, I want to thank Sherry Marshall and Eileen Castle, who were – and are to this day – great sources of inspiration, showing me the excitement in learning new things and the joys of sharing them with others.

I must also thank my mother, Debbie, for her constant encouragement and unflinching willingness to talk about anything and everything that may be on my mind – day, night, or any hour in-between. Equally important was her never-ending forgiveness when I'd, instead,

forget to call. I thank my sister, Mary, for being an incredibly helpful and unbiased guide, having previously navigated the choppy waters of graduate study herself. I look up to her for her ability to conquer anything in her path and giving me confidence that maybe I can too. I thank my brother, Alan, who always let my childhood self tag along while he tinkered and worked in the garage, including, to our parents' dismay, disassembling parts of the lawnmower. He has been a model of determination and balance, reminding me of the importance of seeking out a little island-time attitude now and again.

I must especially thank my wife, Caitlin Cadieux, for her continued support, personal sacrifices, love, and patience through these years of long weeknights and weekends. She's been my closest confidant, my best friend, and I'm lucky to have had her to help me across the finish line.

Finally, I would like to dedicate this work to my late father, Dwight, who fostered my interests in math, science, and astronomy as far back as I can remember, always pointing upward to show me something in the stars.

ABSTRACT

When designing the architecture for a supercomputer, there are many facets of design to consider. Among them, and possibly most important, is the choice of communication network that interconnects the thousands of processors together. The selected communication network forms the backbone of the system, allowing for massive scale parallelism and inter-process coordination. Different patterns of interconnection, or network topologies, have different strengths and weaknesses.

The cost of building a supercomputer is often a significant factor that influences the choice of network topology. Prospective system builders aim to get the highest level of expected performance given their budget and poor or ill-informed choices in system design can be very costly mistakes. Thus, having reliable predictions of how different communication network topologies behave is a critical step in system acquisition. Simulation allows for the rapid testing and procurement of expected performance metrics of full-scale networked systems without needing to physically build them or compromise testing scale.

Network topologies connect switches and any attached compute nodes to each other forming paths of communication from one endpoint to another. As compute nodes inject traffic into the network, packets containing the contents of communication will be routed from one switch to another until finally reaching their destination. With increased levels of traffic, switches may become overburdened, receiving more packets at a rate faster than what they are able to process and route. This imbalance will mean that any packets traversing the overloaded switch will become delayed as they sit in a queue in the switch's memory waiting to be routed.

A switch becoming overloaded is a point of local congestion. Eventually, if the situation remains unresolved, the buffer space on the switch will become full and cannot receive any more packets until another already in its memory is routed away. If other switches have packets destined for the overloaded and full switch, then they may find themselves waiting to forward packets and, consequently, their buffer space begins to fill up. The local congestion previously found on a single switch begins to spread to other nearby switches and the problem worsens, interfering with many more packets and resulting in poor application performance.

Network topologies can be designed to be more resilient to the effects of network congestion. For example, having a high diversity of possible paths between any two endpoints

can provide more alternative routes for packets should one become congested. Clever routing schemes to more effectively balance load across the network or to route around observed points of local congestion can work to mitigate the effects of congestion and thus minimize packet interference. In this work, I look to study, through effective simulation, situations for the occurrence of congestion as well as technologies and methods to mitigate and resolve it.

This document provides an overview of various methods for the avoidance of congestion, including the usage of adaptive routing, quality-of-service techniques, and network topology design. Additionally, it also explores two techniques for the mitigation and treatment of congestion through detection, causal identification, and abatement strategies. Lastly, this document proposes new techniques for effectively simulating large parallel discrete event simulations with simultaneous events – such as network simulation – and demonstrates how it can be used to gain deeper insight into the characteristics of simulated models.

CHAPTER 1
INTRODUCTION

Picture in your mind your daily commute: your hands gripping the steering wheel, your foot near the pedals. Imagine how comfortable – or uncomfortable – your car seat is. This may be easier to imagine for some but is a daily reality for many. In fact, on average, Americans who live in metropolitan areas spend around 27 minutes en route to work daily – one way [1].

One factor that affects how long it takes to get somewhere is distance. Intuition gained from living our lives would tell us that the further away something is, the longer it will take to get there. Simple physics informs us that the time of travel is the distance traveled divided by the average speed so, clearly, distance plays a large factor. However, the average speed plays an equally proportional part in determining how long it takes to get anywhere. In the case of commuting by car, a major limiting factor of vehicular speed is – apart from speed limits – interference of other cars encountered along the route.

It is easy enough to estimate how long it would take to get somewhere by looking at all of the speed limits along the way and creating a weighted average based on the distance that each speed limit is enforced over. This, however, is not realistic; other people exist and are also driving their cars to work, the market, the bank, or even just for a stroll.

How much time it takes to travel from your home to your place of work on empty streets is really only good for establishing a lower bound, a minimum amount of time spent traveling. Typically, however, one does not care how long their commute would take at 3:00 in the morning; they care how long it takes at 8:00 AM and 5:00 PM when there are lots of other cars also driving about.

Similarly, effectively understanding the expected performance of supercomputer communication networks is more complicated than just factoring in the switching speed of its routers or the bandwidth and latency expectations of its links. It relies on considering the effect that packets being transmitted across these switches and links have on each other. Every packet injected into a network incurs a cost to the network: time spent processing, waiting, or being transmitted across links. During its lifetime, a packet will find itself in several memory buffers at various locations in the network and, thus, taking up space where another packet might *want* to be. A packets place in the queue will be directly impacted by the series of decisions that led the packet there as well as those decisions that led all the

other packets in the queue to the same place.

1.1 Competition of HPC Network Resources

Network resource contention, or the phenomenon of when two packets are vying for access to the same system resources, will affect the resulting path and time of travel of packets traversing the network. The effect that said contention has on packets, and by extension the processes that they facilitate, is called packet interference. Like packets on a network, other cars that you encounter during your commute, ahead of you in the traffic signal queue waiting to turn left at the unprotected green light, are interfering with your commute and affect decisions you make while driving and your overall transit time.

A large focus of High-Performance Computing (HPC) research is the aim of finding the correct communication network and associated technologies to maximize the expected system performance while minimizing the cost of acquisition and construction. HPC architecture design has been reaching toward the threshold of exascale computation performance: conceiving a supercomputer that can facilitate at least one quintillion floating-point operations per second (1 exaFLOPS).

To breach the exascale threshold requires innovation across several aspects of design. While processor performance is a significant factor when it comes to the power of a system, the communication interconnect – the network of switches carrying packets of application inter-process communique – can make up a large portion of the overall system cost and determine many limiting factors such as size and maximum performance.

Packets are injected into this network from endpoints known as compute nodes or terminals. Each packet is destined for another compute node but must first be routed through the interconnection network via switches or routers.

A trivial way to minimize system time spent during communication and thus maximize the time spent performing meaningful computation would be to connect each compute node to each other in an all-to-all fashion. However, as bigger systems with more compute nodes are built this type of interconnection scheme becomes incredibly expensive, if possible at all, due to hardware limitations, as the number of links required grows significantly faster as we increase the size of the network. In order to maximize performance while minimizing cost, clever strategies in connecting switches must be employed.

Different patterns of interconnection are known as network topologies. Varieties of

topologies have different strengths and weaknesses, but the main metric for comparing between them is their level of expected performance in relation to their associated cost. A strong performing topology may not be favorable if its costs are significantly more than others or if it is not scalable to larger system sizes. Conversely, a low-cost topology may not be desired if it also lacks in performance compared to others in its class.

While the shape of a network interconnect strongly influences the communication capabilities, there are other aspects of system design that can affect how well the network tolerates high levels of injected traffic. A frequent occurrence in High-Performance Computing (HPC) communication networks is packet congestion. This happens when packets become backed up at switches, waiting to be routed to the next hop toward their destination. If congestion is left unchecked, it can spread and cause system-wide congestion and lower performance.

1.2 HPC Network Topologies

Modern-day HPC network designs come in a broad range of shapes and sizes. One of the more commonly used topologies is the Fat-Tree network [2]. This network shape forms a tree-like structure where aggregate link bandwidth becomes higher the closer to the root level the links become. These networks are highly scalable and have been utilized in some of the world's most powerful supercomputers of their time, including the Department of Energy's Summit and Sierra [3], [4] supercomputers, as well as Rensselaer Polytechnic Institute's own AiMOS supercomputer [5]. At the time of writing, the most powerful supercomputer in the world, Supercomputer Fugaku, utilizes a Torus-based design known as the Tofu Interconnect D [6]. Interestingly, torus-based designs featuring rectilinear arrays of multiple dimensions and linking degrees were by no means "new" when Fugaku was brought online in 2020. Torus designs were featured in the Cray T3D network [7] and IBM's Blue Gene line of supercomputers [8]–[12].

This document presents work focusing mostly on a class of network topologies known as Dragonfly [13]. Originally designed as a network of switches clustered into groups to form a large, high-radix "virtual router," the idea was to create an incredibly scalable network. Each group consisted of an all-to-all local connection of routers with global links to other groups as well. Each group would have at least one global link to each other group in the network. This is, essentially, the 1D-Dragonfly topology shape which is shown in Figure 1.1. This

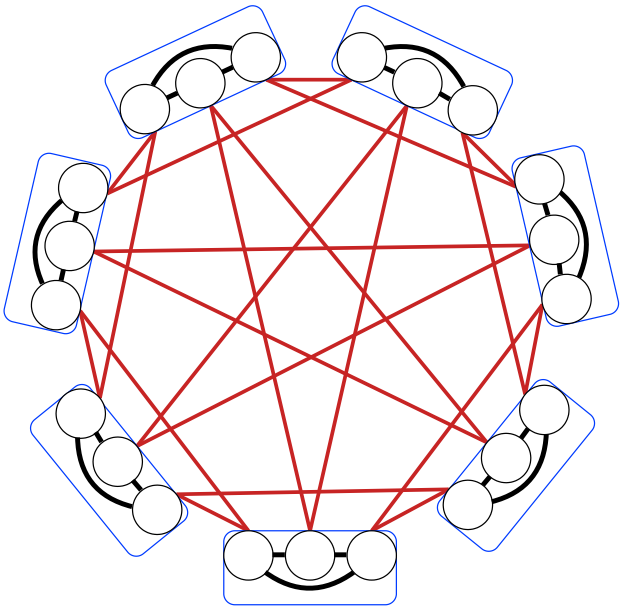


Figure 1.1: Diagram of an example 1D Dragonfly switch topology. Switches in each local group are connected to each other in an all-to-all fashion. Additionally, each group is connected to each other in an all-to-all pattern with at least one connection between each pair of groups.

design inspired numerous other network topologies including the 2D-Dragonfly, Dragonfly Plus/Megafly, and Slim Fly.

The 2D-Dragonfly differs from its one-dimensional counterpart by utilizing a mesh grid within each local group as opposed to an all-to-all interlinking. This design was featured in the Cray Cascade HPC system [14]. Another variant, Dragonfly Plus [15] (also known as Megafly by some), features a complete bipartite graph within each local group, delegating global link responsibilities to one-half of a group's switches and compute node terminal responsibilities to the other half. The last type of dragonfly-esque networks featured in this work includes the Slim Fly topology [16], which features a complex mathematical connection generation scheme that guarantees a low network diameter.

1.3 Managing Congestion

Different routing algorithms can be applied to switches to route packets to try and minimize the risk of congestion or to mitigate its effects if it already exists in the network. For best communication performance, on an empty network, the best choice of routing a packet is a minimal (MIN) path which guarantees a minimum number of hops from one

endpoint to another. As more and more packets attempt to take the minimal path through the network, hotspots of traffic may form, and packets begin competing for resources on switches as they are routed through the network.

In order to reduce the likelihood of hotspots, a different routing algorithm can be applied: non-minimal valiant routing (VAL) [17]. This type of routing picks a random intermediate router that a packet must route to first before being routed to its final destination. This spreads traffic out across the network, reducing the likelihood of there being any one spot on the network that has significantly more traffic at any given time. This type of load balancing comes with a cost, however, in the form of increased packet end-to-end latency. Packets must be routed along several links to some other destination in the network before making their way to their desired endpoint, incurring some time-cost at every hop in its path. Other routing algorithms, referred to as adaptive routing, may try to route minimally when possible but will pick intermediate routes to avoid hotspots should they be discovered.

Aside from clever routing, other techniques for preventing congestion can also be applied. Quality of Service (QoS) techniques are well known in TCP/IP networks [18] but are relatively new to the HPC research space. Applying QoS to a network interconnect can put restrictions on the total amount of bandwidth used by different designations of traffic known as traffic classes. Packets can be classified into different traffic classes based on various criteria. In our work studying QoS on HPC systems, we apply two different classification methods, one for giving priority to packets belonging to specific jobs and another for giving priority to packets carrying messages participating in latency-sensitive, broad-participation, collective operations.

Resource contention and inter-job interference [19], a consequence of congestion, is most likely to occur when two packets find themselves in the buffer space of the same router. They each require time by the switch's processor to route and forward the packet toward its destination and are competing for switch resources. If we increase the number of routers in the system without increasing the average hops between any two endpoints, then we reduce the likelihood of two packets "interacting." A simple-to-understand method to accomplish this is in multi-planar networks: networks with more than one independent plane of routers for traffic to be injected onto. Work has been performed to study the benefits of multiple planes of independent routers on various network topology designs.

In efforts to better understand how to manage congestion, it is important to understand

the circumstances that can cause congestion. One of those circumstances is link failure: when a link connecting two switches in the interconnect can no longer transmit packets. Higher levels of link failure can reduce the path diversity in the network, limiting the number of viable paths between two endpoints. This makes hotspots more likely and, consequently, congestion. We introduced work that studies the resilience of networks of varying configurations to link failure. Network types that did not handle link failure as well had poorer application performance due to congestion and increased communication time.

1.4 Simulation of Interconnection Networks

Physically building an HPC test bed to the specifications necessary for system builders to make confident decisions is often cost-prohibitive. While analytical methods can be employed to predict steady-state behavior, it is often easier to model and simulate interconnection networks instead - especially at full-sized scales.

Simulation carries its own challenges, however. Accurately measuring performance metrics on varying network models with multiple workloads and configurations in an effective manner means striking a balance between coarse and fine granularity. The more detail that a simulation has, the more complicated the model and the more technical overhead required to run it. A balance between precise, accurate results and runtime must be struck.

To achieve as fine granularity as possible, we leverage the parallelism of Parallel Discrete Event Simulation (PDES) to maintain acceptable levels of simulation runtime given the complexity.

1.4.1 CODES

The CODES (Co-Design of Exascale Storage) simulation framework provides high-fidelity, massively parallel simulations of prototypical next-generation HPC architectures. CODES was originally created to enable the easier development of HPC network and storage system models [20]. The framework has been extensively used for performance analysis of modern interconnect topologies (fat tree, torus, dragonfly, express mesh, and slim fly) [21]. The network models have been validated against real architectures [22]. CODES has numerous modules and models with a wide variety of configurable features within each model, including modeling of high-performance computing storage systems [23]. A general overview of CODES' parts can be found in figure 1.2.

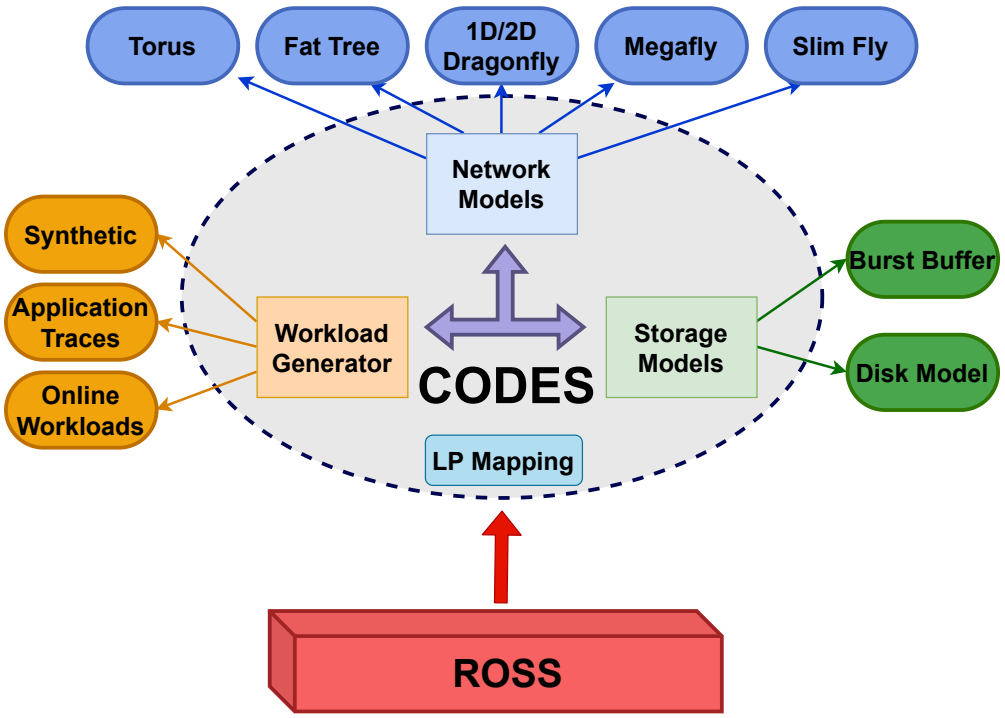


Figure 1.2: Overview of the CODES interconnection network simulation platform.

A simple CODES simulation consists of a network model, a traffic-generating workload, and a configuration that specifies the details used to set up the simulation.

With CODES, users can evaluate the features and behavior of network models with varying configurations and HPC workloads. The feature set of CODES is wide-ranging. It includes data collection services, additional PDES features such as a reverse computation stack to simplify complex simulation rollback behavior, and tools for generating synthetic traffic as well as replaying HPC application traces. The supported traffic generators, known as workloads, can be run on a given network independently or simultaneously with arbitrary endpoint allocation mapping to help simulate real-world usage and performance.

1.4.2 ROSS: Parallel Discrete Event Simulation Engine

We utilize ROSS [24] as the core simulation framework. ROSS, a PDES engine, abstracts a simulation into logical processes (LPs). Each LP represents an entity in the simulation, for example, a network switch, a network terminal, or a workload server that generates traffic.

In ROSS, any interaction between LPs is considered an *event*. Every event has a timestamp representing the time within the global simulation that the event occurred.

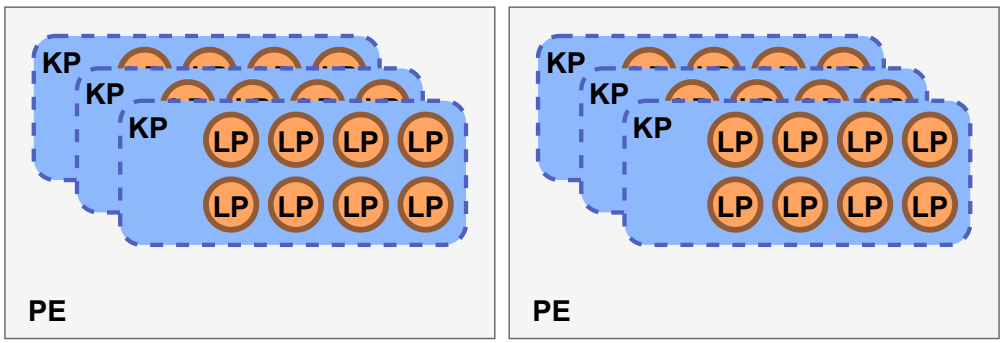


Figure 1.3: Visualization of the ROSS simulation structure hierarchy.

The set of all LPs is divided among processing elements (PEs), which process events scheduled by their respective LPs in timestamp order. There is one PE per physical MPI rank for the execution of the simulation. Within each PE is an additional organizational abstraction called kernel processes (KPs). KPs are a sort of container for LPs and aid in the organization and synchronization of LPs during simulation. A visualization of this hierarchy is found in Figure 1.3

ROSS allows for sequential, conservative, and optimistic execution. *Sequential* execution maps all LPs to a single PE, and events are deterministically executed in exact timestamp order. Since all LPs process and schedule events using the same clock, causality is never infringed upon.

Conservative execution [25] is a type of parallel execution that has additional synchronization overhead to prevent events from ever being processed out of order across all PEs. Because of this layer of synchronization, the causality of events is always correct in this mode. This can be enabled through the usage of what is referred to as the Chandy-Misra-Bryant algorithm utilizing Null Messages to prevent simulation deadlock [26].

Optimistic execution allows each LP to schedule events according to its own local clock. Therefore, PEs could process events in an order that disagrees with causality on other PEs. Given relatively infrequent causality errors, optimistic execution is faster than conservative since the latter has a much higher synchronization overhead. The performance of optimistic execution is hindered by the frequency of causality errors because these must be handled before the simulation can progress. ROSS handles causality errors through reverse computation [27] as an optimization of the *Time Warp* protocol [28].

When out-of-order events are detected, all events resulting from such a conflict must be undone, in other words, reverse computed, until the state of the simulation is what it

was just before the incorrect event was scheduled. Each LP is programmed by the model developer with a reverse-computation handler that gives the LP the functionality required to undo an event. This reverse handler typically just undoes any LP state changes made by the forward event handler.

ROSS gives each LP its own set of random number generators (RNGs) that can be independently rolled back any number of times to deterministically reproduce the same sequence of random numbers after being rolled back.

A simple example of a ROSS PDES simulation is a toy random-packet-routing model with each LP in the simulation representing a single router. Routers forward packets to each other randomly through *events* and keep track of how many packets they received. When the simulation is forward progressing, router LPs will receive a packet from a neighboring router. They must then (1) increment its received packet count, (2) randomly pick a new neighbor to forward the packet to, and (3) schedule and send the packet in a new event to the chosen neighbor. When ROSS detects that an event was processed in an order that conflicts with causality, router LPs then have to undo the conflicting events by (1) decrementing its received packet count; (2) rolling back the RNG used to pick a neighbor; and (3) sending an anti-message to the previously chosen neighbor, signifying that the resulting message was sent in error.

1.5 Workloads

To simulate how different configurations of interconnection networks behave, we need to inject packets of data into the network to be routed. CODES supports several types of workloads out of the box, and extensions have been created that expand the supported set. The function of any CODES workload LP is to create messages destined for other workload LPs to be injected as packets into the simulated network and routed to the correct destination.

The CODES framework maps MPI workload LPs, each representing the MPI ranks that would be operating on physical compute nodes of a given network, to *terminal node* LPs in the network model. The workload LPs operate as the origin and final termination of messages in the network. When a workload LP generates a message, CODES passes the message to the mapped terminal node for injection into the network.

In this work, the workloads used fall into three categories: synthetic, HPC application traces, and online workloads.

1.5.1 Synthetic Workloads

With CODES, any arbitrary pattern of communication can be created by developing a synthetic traffic generator LP. Synthetic workloads [29] are, generally, simple patterns that are easy to describe (and thus easy to program). Examples include patterns such as uniform-random, random-permutation, ping-pong, broadcast, and many-to-one (also known as incast). These types of workloads are helpful for creating "dummy" traffic in the network, filling switch buffers with traffic to interfere with the primary workload of study.

Synthetic workloads are also helpful for stress testing simulated networks. By configuring each synthetic workload LP to generate as much traffic as can be injected by its associated network terminal, the network can become saturated, and its effective throughput measured.

Because of the flexibility, synthetic workloads are very helpful as diagnostic or sanity checking tools. It is important, however, to recognize the simplicity of this type of workload - the generated traffic does not exactly represent any larger pattern of communication one would expect from a real-world HPC application without explicit modeling of that behavior. Thus, a synthetic workload can provide helpful insight into the performance of a simulated network but cannot easily provide the whole picture of how a simulated system may behave in the real world.

1.5.2 HPC Application Traces

Stepping toward more realistic workloads, we have the trace-driven workload type. These workloads use HPC application communication traces captured from real-world running applications. Traces generally come in DUMPI MPI trace format [30], which lays out the specific details of MPI messages sent during the source application.

Unless otherwise noted, the network traces used in this work are provided by the Design Forward Program [31]. The specific workloads used are noted in each chapter.

1.5.3 SWM Online Workloads

Another step toward realism comes with the introduction of online workloads into the CODES simulator. Intel's Scalable Workload Model (SWM) workloads generate traffic on-the-fly, like synthetic workloads, but does so based on the behavior of a skeleton application simulating the actual computations performed by full real-world applications. They are a workload representation approach that focuses on representing the communication patterns,

dependencies, computation-communication overlap, and algorithms. They are flexible; they can be configured in size and intensity to provide a useful tool for testing networks and extracting reliable predictions of network performance.

The SWM code is decoupled from the original application code as well as from any particular simulator, enabling use across different simulation environments. The SWM runtime supports a set of low-level API communication primitives to support a number of MPI-based communication operations. The primitives used by the SWM closely resemble those of MPI and SHMEM, but they are not constrained to specific syntax or semantics.

As an added benefit, they can be used to simulate large, high-intensity workloads without needing extremely large DUMPI trace files. As a result, SWMs provide an avenue for a significant reduction in memory required for simulation.

In this work, we utilize several SWM representations for multiple HPC codes, including Nekbone, LAMMPS, MILC, nearest neighbor, and two newly created patterns: Fixed Pairs and PeriodicAggressor [32].

1.6 Summary of Contributions

Following this introduction, we have three main parts containing work published in HPC and Simulation focused venues discussing the following contributions:

1. Developed and evaluated CODES Megafly network model with MIN, VAL, and PAR routing. This is featured in Chapter 2.

2. Developed and evaluated Quality of Service module for CODES 1D Dragonfly and Megafly models. This is featured in Chapter 2.

3. Developed and evaluated Multi-Rail-Multi-Planar Configurations of the CODES Slim Fly model. This is featured in Chapter 3.

4. Developed and evaluated Multi-Rail-Multi-Planar Configurations of CODES 1D Dragonfly and Megafly models. This is featured in Chapter 4.

5. Developed module for enabling and configuring link failure in CODES 1D Dragonfly and Megafly, used module for evaluation of single and dual planar network resilience to increasing levels of link failure. This is featured in Chapter 4.

6. Designed and developed a centralized mechanism for the detection, causal identification, and abatement of broad-scale congestion in an HPC communication network. This is featured in Chapter 5.

7. Designed and developed a distributed mechanism for the detection, causal identification, and abatement of precise, localized congestion in an HPC communication network. This is featured in Chapter 6.

8. Present three mechanisms for biased and unbiased deterministic ordering of simultaneous events in a Parallel Discrete Event Simulation with and without zero-offset events. This is featured in Chapter 7.

9. Exhibited impacts of unbiased deterministic ordering of simultaneous events in the context of HPC interconnect simulation with CODES. This is featured in Chapter 8.

In Part I, we explore the effects of congestion in HPC communication networks as well as ways to avoid them. Specifically, in Chapter 2, we explore the use of Quality of Service techniques on the 1D Dragonfly and Megafly networks. We show how congestion from inter-job packet interference can cause lowered application communication performance and demonstrate two traffic-class classification techniques and their ability to mitigate network resource contention and the effects of congestion.

In Chapter 3, we present Fit Fly, a multi-rail-multi-plane configuration of the Slim Fly network. This type of network has additional planes of routers, independent from the first and other planes, where packets can be injected into the network from a shared pool of terminals. The additional planes of routers give additional capacity to the network and reduce the likelihood of any two packets finding themselves on the same router buffers - thus reducing resource contention. While this type of network has considerably increased cost, the positive benefits are maintained with cheaper, reduced bandwidth, hardware.

Building on those findings, Chapter 4 extends the 1D Dragonfly and Megafly network models to also support multi-rail-multi-planar configurations. This chapter also introduces the capability of CODES to simulate network link failures, which comes with its own challenges. We evaluated the resilience of multiple 1D Dragonfly and Megafly network configurations, single and dual planar, by performing a link failure study. The findings from Chapter 3 are maintained as the dual-planar configurations had considerably improved performance even

with significant levels of link failure. This further supports the argument that the increased cost of multi-rail-multi-planar networks may be worth the significantly increased performance and resiliency that comes with it.

Part II looks into how congestion can be managed in a network once it is known to exist. Chapter 5 presents a method for detection, causal identification, and abatement of congestion in a 1D Dragonfly network. This chapter takes a broad approach toward detecting broad, widespread congestion and attempts to identify and quell the cause. In Chapter 6, an improved, more precisely targeted method is presented, which works to reign in congestion causing aggressor workloads before congestion becomes widespread so as to minimize the impact on other workloads in the system.

Part III discusses the need for effective simulation and its impact on the study of HPC communication networks. Chapter 7 posits three strategies for creating a deterministic total ordering from a partial ordering of simultaneous events in a parallel discrete event simulation. These strategies find both biased and unbiased random orderings of events without violating causality with and without the existence of zero-offset events. Chapter 8 adapts the CODES dragonfly network model to utilize this newly developed feature of ROSS and explore its impact on ease of development, removing sources of bias, and analysis of results.

Finally, Part IV presents related work in Chapter 9 and makes concluding remarks in Chapter 10.

PART I

CONGESTION AVOIDANCE

CHAPTER 2
EVALUATING QUALITY OF SERVICE TECHNIQUES ON THE MEGAFLY NETWORK

An emerging trend in High-Performance Computing (HPC) systems that use hierarchical topologies (such as dragonfly) is that the applications are increasingly exhibiting high run-to-run performance variability. This poses a significant challenge for application developers, job schedulers, and system maintainers. One approach to address the performance variability is to use newly proposed network topologies such as megafly (or dragonfly+) that offer increased path diversity over a traditional fully connected dragonfly.

In this work, we select HPC application workloads that have exhibited performance variability on current 2-D dragonfly systems. Using the Scalable Workload Model (SWM) library and CODES HPC systems simulation framework, we evaluate the baseline performance expectations of these workloads on the megafly and 1-D dragonfly network models. Our results show that in the majority of the cases, a megafly network is more resistant to communication interference than a fully connected 1-D dragonfly network. However, in order to sufficiently mitigate the performance variability, additional quality of service (QoS) levels need to be explored. We use bandwidth capping and traffic differentiation to introduce multiple traffic classes in megafly networks. In some cases, our results show that QoS can completely mitigate application performance variability while causing minimal slowdown to the background network traffic.

2.1 Introduction

With modern high-performance computing (HPC) systems shifting to hierarchical and low-diameter networks, dragonfly networks have become a popular choice. They have been deployed in multiple high-performance systems, including Cori, Trinity, and Theta systems at NERSC, Los Alamos National Laboratory, and Argonne National Laboratory, respectively [33]–[35]. Dragonfly is a hierarchical topology that uses short electrical links to form groups of routers using a 1-D or 2-D all-to-all interconnect. These groups are then

Portions of this chapter previously appeared as: M. Mubarak, N. McGlohon, *et al.* "Evaluating quality of service traffic classes on the megafly network," in *Proc. Int. Conf. High Perform. Comput.*, 2019, pp. 3–20.

connected all-to-all via optical links. While this design offers low diameter and cost, it increases contention for the link bandwidth among multiple applications, which introduces performance variability [36]. For next-generation exascale systems, HPC designers are considering variations of the dragonfly topology that offer increased path diversity, fairness, and scalability [15]. One such topology that has been recently proposed is the dragonfly+ – or megafly – network, which uses a two-level fat tree to form groups of routers. These groups are then connected all-to-all via optical links. Megafly networks use the path diversity of a two-level fat tree to alleviate the communication bottlenecks that can be introduced with standard dragonfly networks. Megafly networks also have the added advantage of using only four virtual channels (VCs) for deadlock prevention (including request and response traffic) as opposed to eight virtual channels used in a fully connected 1-D dragonfly network. Prior work [37] has shown that the design of megafly networks helps mitigate performance variability to some extent, it does not completely eliminate it.

Although quality of service (QoS) has been investigated and implemented on TCP/IP networks and data centers [18], the mechanism remains largely unexplored in the context of HPC networks. In the past, HPC network topologies, such as the torus, had mitigated communication interference by dedicating isolated partitions to individual jobs. Introducing QoS traffic classes can be a useful way to mitigate interference on the now popular hierarchical networks. In this work, we use HPC application workloads that demonstrate performance variability on current dragonfly systems as shown by Chunduri et al. [36]. We replay them on CODES packet-level interconnect simulations [21], [38] to answer questions about dragonfly and megafly network topologies: How does the performance of a megafly network compare with a fully connected dragonfly network? How do traffic classes help with performance variability on a megafly network?

The contributions in this chapter are as follows:

1. We evaluate the performance variability of HPC application workloads on both a megafly and a 1-D dragonfly network using similar network configurations. We compare the performance of a megafly network with a 1-D dragonfly to determine which is more resistant to perturbation.

2. We exploit the fact that megafly requires fewer virtual channels for deadlock prevention (as compared to conventional dragonfly), and we use the unused VCs to introduce QoS

traffic classes. We evaluate two mechanisms through which QoS can be introduced in HPC networks. First, using bandwidth capping and traffic prioritization, we quantify the impact of QoS when an entire high-priority traffic class is dedicated to an application or set of applications. Second, we dedicate the high-priority traffic class to latency-sensitive operations such as MPI collectives and observe the performance improvement.

3. We extend the CODES simulation framework to perform packet-level simulation of HPC networks in an online mode driven by the scalable workload models (SWM) [32] for use in the above-mentioned experiments.

2.2 Exploring Quality of Service on HPC Networks

In the past, HPC systems were often constructed with torus networks, and jobs were allocated onto partitions of the network that reduced resource sharing and communication interference. With hierarchical networks sharing resources such as switches and links, partitioning becomes more difficult and introducing traffic classes becomes an important step to mitigate communication interference. For example, Figure 2.1 shows the performance degradation of two HPC communication workloads on a model of Argonne's Theta Cray XC system, which has a 2-D dragonfly architecture. The performance modeling is done using the CODES simulation framework, and the 2-D dragonfly network simulation in CODES has been validated against Argonne's theta network using synthetic communication benchmarks [22]. The background traffic injection rates are at a percentage of the maximum link capacity, and the application slowdown increases with more background injection. The slowdown in communication time can significantly impact the overall application performance as the typical range of communication time in communication-intensive applications is in the range of *50-80%* [39]. Performance variability along the same lines has been reported in [36], where the actual system is seeing a slowdown of up to 2x in overall performance.

Current 2-D dragonfly networks use up to 8 virtual channels to prevent deadlocks, which is typically all the VCs available. Megafly networks use only 4 VCs for deadlock prevention (2 VCs for request traffic and 2 for response traffic), thus making them a better candidate for enabling multiple traffic classes. In this chapter, we explore quality of service on megafly networks based on traffic prioritization and bandwidth shaping [41]. Figure 2.2 shows one way to implement quality of service on HPC networks. In this implementation, a

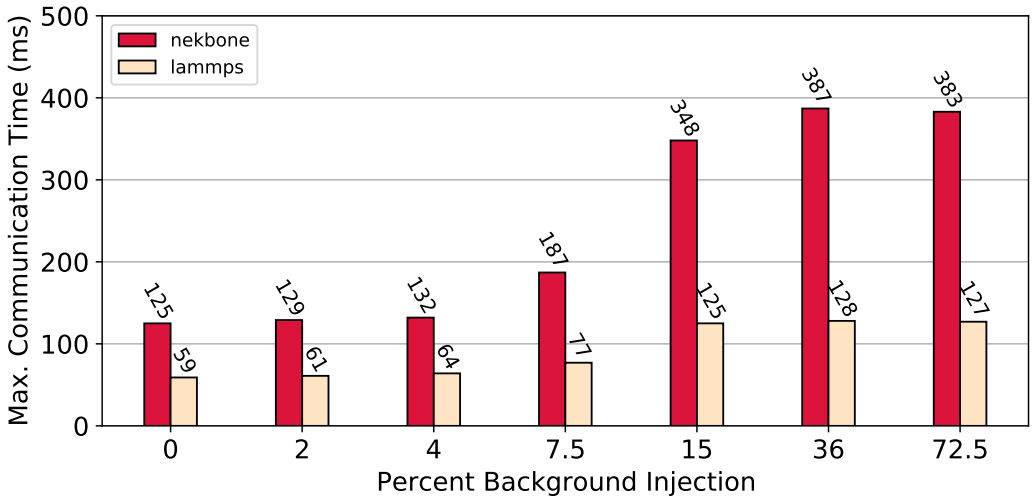


Figure 2.1: Communication time slowdown of LAMMPS (1,024 ranks) and Nekbone (1,000 ranks) with background traffic on a Theta network model with 3,456 network nodes. Uniform random workload with large messages is used for background traffic. Background traffic is injected relative to maximum link bandwidth (x-axis shows the percentage of link capacity consumed). Data sourced from [40].

bandwidth monitor component in each switch tracks the bandwidth consumption of each traffic class for every port. The bandwidth monitoring is done over a static time window t_w. Each traffic class is assigned a certain fraction of maximum available link bandwidth, which serves as the upper bandwidth cap for that traffic class while the link is oversubscribed. If the bandwidth consumption of a traffic class reaches the cap and the link is oversubscribed, the traffic class is designated as inactive for the remaining duration of the static window t_w. An inactive traffic class has the lowest priority and it gets scheduled only if there are no packets in the remaining higher priority traffic classes. At the start of the window t_w, the bandwidth statistics for each traffic class are reset to zero, and the traffic class(es) marked as inactive are activated again. If all the traffic classes are violating their bandwidth cap, then a round-robin scheduling policy is used for arbitration.

The implementation of QoS can be beneficial in reducing communication interference on hierarchical networks. For instance, a common problem exhibited on such networks is that communication-intensive (or bandwidth-hungry) applications can "bully" less-communication-intensive applications [38]. With QoS-enabled networks, bandwidth-hungry applications can be prevented from exceeding their permissible bandwidth limits. This approach allows

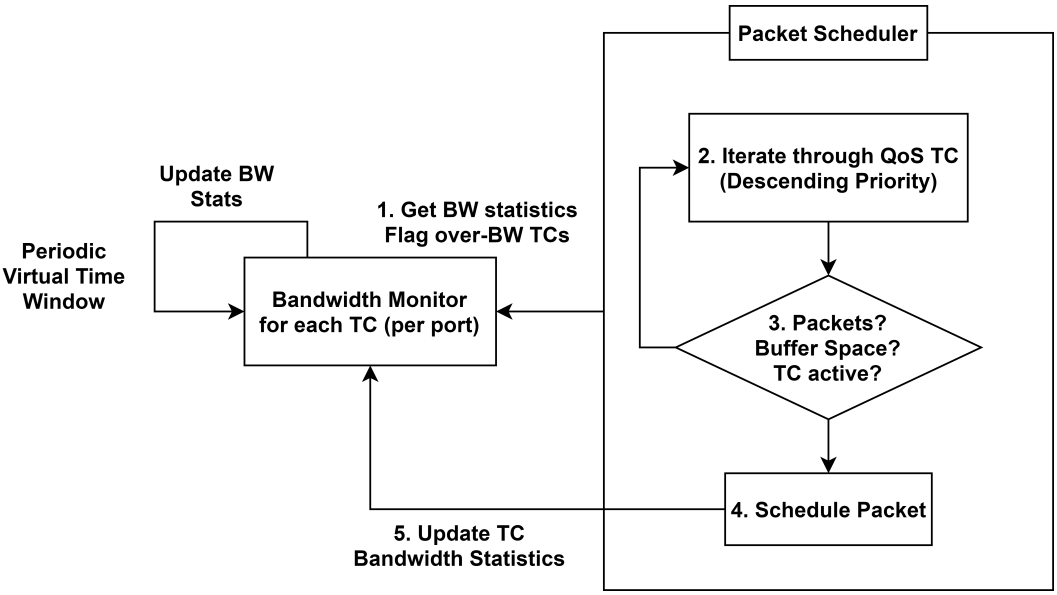


Figure 2.2: Enabling quality of service on HPC networks (TC – traffic class, BW – bandwidth, QoS – quality of service).

less-communication-intensive applications to have their fair share. Alternatively, one can assign a high-priority traffic class to latency-sensitive operations such as MPI collectives. We report on experiments with both of these QoS mechanisms in Section 2.6.

2.3 Integrating SWM with CODES

Prior to this work, the CODES simulation framework supported system simulations with post mortem communication traces. Although traces can illustrate realistic system behavior (for a given problem size), their use inhibits flexibility and simulation scalability as compared to other workload representations. Therefore, we extended the CODES simulation suite to replay workloads in an online or in situ mode using the Scalable Workload Models (SWMs) presented in [32].

To avoid storing large communication traces, we enabled in situ workload replay with SWM through the Argobots library [42] to synchronize between CODES and SWM workloads. Argobots is a low-level threading framework that is used to synchronize the execution of the SWM workload in parallel with the CODES simulation framework. More details on CODES and SWM integration using Argobots are discussed in [43].

2.4 Evaluation Methodology

In this section, we discuss the network configurations, workloads, rank-to-node mapping policies, and routing algorithms used in the study.

2.4.1 Topology and Routing Description

The dragonfly network topology, proposed by Kim et al. [13], consists of groups of routers that are connected to each other with one or more optical channels. Within each group, the routers are directly connected to each other in an all-to-all manner via electrical links. In this chapter, we refer to this configuration as a 1-D dragonfly. A variation of a dragonfly topology, deployed in the Cray XC systems, uses a 2-D all-to-all within each group instead of all-to-all connections. We refer to this configuration as a 2-D dragonfly. A 2-D dragonfly traverses almost double the number of hops as a 1-D dragonfly. The hop count traversed by a 1-D dragonfly is close to that of the megafly network, therefore, we compare megafly with a 1-D dragonfly to ensure a reasonable comparison. Various forms of adaptive routing have been proposed for dragonfly, which detect congestion and determine whether the packet should take a minimal or non-minimal route.

Routing in a dragonfly network is done by taking either a minimal (typically direct) or nonminimal path. A minimal path uses the global channel connecting the two groups with the source and destination nodes. With a nonminimal path, Valiant's algorithm [17] is used to route a packet on a minimal path to a randomly selected intermediate router and then route the packet minimally to the destination router. Nonminimal routes are taken if potential minimal routes are congested. Various forms of adaptive routing have been proposed that detect congestion and determine whether the packet should take a minimal or nonminimal route. We use the progressive adaptive algorithm (PAR) provided in [44]. The PAR algorithm in the simulation re-evaluates the minimal path until either the packet decides to take a nonminimal route or the packet reaches the destination group on a minimal path. A 1-D dragonfly network uses up to 6 virtual channels to avoid deadlock, 3 for request and 3 for response. However, prior work shows that three virtual channels for either request or response traffic can cause congestion in the intermediate group and suggests adding a fourth virtual channel, which is helpful in alleviating the congestion [44]. In this work, we use eight virtual channels for progressive adaptive routing in a dragonfly network.

What separates various dragonfly topologies from each other is largely based on the

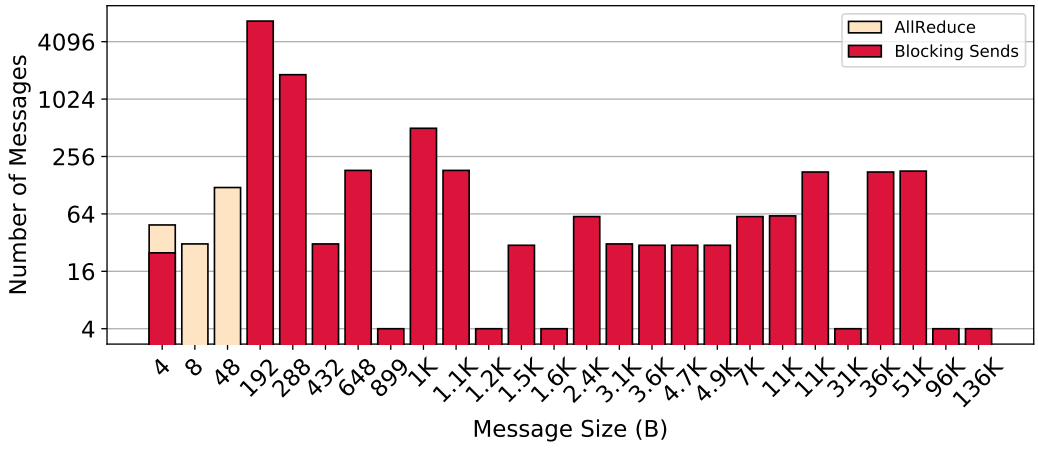
Table 2.1: Configurations of megafly and 1-D dragonfly used for performance comparison.

	Megafly	1-D Dragonfly
Router Radix	32	32
Groups	33	65
Nodes/Group	256	128
Node Count	8448	8320
Global Connections / Group	256	128
Link Bandwidth	25GiB/sec	25GiB/sec
Nodes Per Router	16 (Leaves only)	8

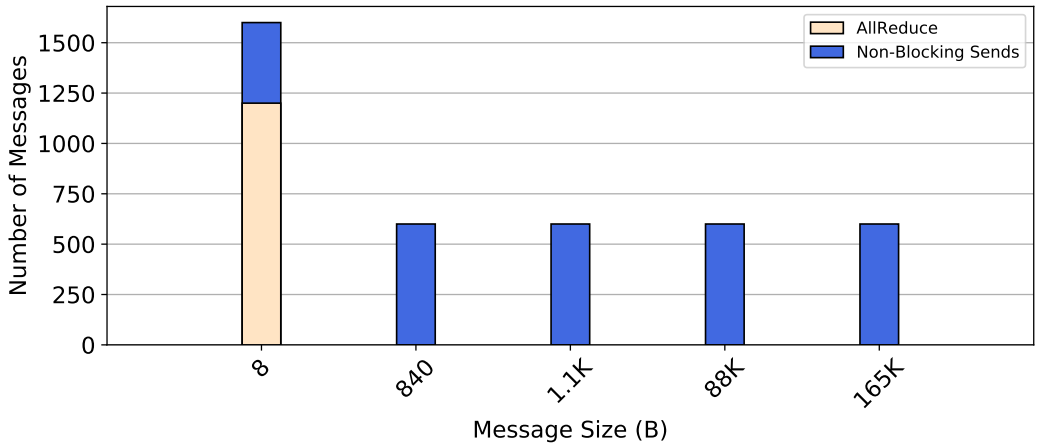
interconnect within a group. Megafly is a topology that belongs to the Dragonfly class of interconnection networks. At a high level, it is classified as having groups of routers that are, in turn, connected to each other with at least one global connection between any two groups. Megafly is characterized by its connectivity in the form of a two-level Fat Tree network in each group. This locally defined network is also known as a complete bipartite graph, a graph with two sub-groups where all nodes within one subgroup are connected to all nodes in the other subgroup. There are no connections between the routers within the same subgroup. The two levels in each group have routers that will be referred to as *Leaf Routers*, those that have terminal/compute node connections but no global connections, and *Spine Routers*, those that have global connections to other groups but no terminal/compute node connections [15], [37]. In this chapter, we use the progressive adaptive routing algorithm proposed in prior studies on Megafly networks [37]. Megafly requires only four virtual channels (VCs) to avoid deadlock and none to avoid congestion in the intermediate group.

2.4.2 Network Configurations

To perform a comparison of the megafly network with a 1-D dragonfly, we maintain similar router radix and similar node counts. We used a router radix of 32 ports for both networks. Across the group, the routers are connected via global channels. The configurations of the 1-D dragonfly and megafly are given in Table 2.1.



(a)



(b)

Figure 2.3: Message distributions for LAMMPS (a) and Nekbone (b) application workloads. Data sourced from [40].

2.4.3 Workloads

In order to quantify the slowdown of a particular job due to communication interference, multiple jobs need to be running in parallel to exhibit interference. We conduct two types of interference experiments: (i) replay HPC applications that serve as foreground communication traffic in conjunction with a job that generates synthetic background communication to understand the interference in a controlled manner, and (ii) replay multiple HPC applications in parallel to capture the dynamism of multi-phased communication and quantify the impact of perturbation.

2.4.3.1 *Foreground Traffic*

Previous work demonstrates the performance variability shown by LAMMPS, Nekbone, and MILC applications on the Cray XC40 system [36]. Thus, we use LAMMPS and Nekbone workloads as foreground workloads for our experimental analysis. We also use a 3-D nearest-neighbor communication pattern, which is a commonly used pattern in several HPC applications.

LAMMPS is a large-scale atomic and molecular dynamics code that uses MPI for communication. We use the SWM code that derives its communication pattern from the LAMMPS application. Figure 2.3(a) shows the message distribution of LAMMPS SWM per rank in a problem involving 2,048 ranks. The LAMMPS workload uses MPI_AllReduce with small messages as well as blocking sends and non-blocking receives for point-to-point communication with large messages.

Nekbone is a thermal hydraulics mini-app that captures the structure of the computational fluids dynamics code Nek5000. Nekbone's SWM communication pattern is derived from Nekbone. Figure 2.3(b) shows the message distribution of the Nekbone SWM on a per rank basis in a problem with 2,197 ranks. Nekbone performs a large number of MPI collective operations with small 8-byte messages. It uses non-blocking sends and receives to transmit medium-sized messages.

Cartesian neighborhood communication is a pattern commonly used in multiple scientific applications, including Hardware Accelerated Cosmology Code (HACC), fast Fourier transform solvers, and adaptive mesh refinement (AMR) codes. We use a 3-D nearest-neighbor SWM in this work that transmits large messages (64 KiB and 128 KiB) on a per rank basis with multiple iterations of MPI non-blocking sends and receives followed by MPI_Wait_All. A problem size of 4,096 ranks is used with the nearest neighbor SWM.

2.4.3.2 *Background Traffic*

The background communication traffic is needed to interfere with the foreground workloads. To ensure an even distribution of traffic that covers a significant fraction of the network, we use a uniform random communication pattern. This is generally considered a benign traffic pattern for dragonfly networks. However, with large messages randomly sent in the network, uniform random causes hotspots at multiple network locations and becomes a source of interference. We varied the amount of data transmitted via uniform random traffic

and observed the effect of different data transmission rates on the foreground traffic. The background traffic generation is modeled as a separate job that runs in parallel with the foreground traffic and occupies at least 25% to 50% of the network. The background injection rates depend on the available compute node to router link bandwidth in the network. We inject traffic at a percentage of the available link bandwidth and vary the injection rates between 2% to 36.5% of the link bandwidth. At the 36.5% rate, each node is injecting 9GiB/sec of background traffic with an aggregate network background interference of 18TiB/sec. At this rate, we see significant slowdown (up to *4x* for uniform random and up to *7x* for random permutation) in application communication times for both networks. Therefore, we keep that as the maximum background injection rate.

We experiment with two different background communication patterns: (i) a uniform random synthetic pattern where a rank randomly chooses a destination rank and transmits large messages and (ii) a random permutation traffic pattern where a pair of ranks communicate and transmit data until a certain threshold is reached.

2.4.3.3 *Multiple Applications*

As a specific instance of representative HPC scenarios, we ran the three foreground workloads in parallel (Nekbone, nearest neighbor, and LAMMPS). We also ran each of these workloads in isolation on the network to determine the baseline performance and observed the slowdown introduced when the workloads are running in parallel.

2.4.4 Rank-to-Node Mappings

Ranks are placed on network nodes in a manner similar to that for production HPC systems, where clusters of available network nodes are assigned to a job. Therefore, we use a geometric job placement policy in which multiple clusters of network nodes are assigned to jobs. In the simulation, the clusters are formed by using the inverse transform sampling method for creating random samples from a given distribution. The experiments in the chapter were performed with three different rank-to-node mappings; however, there was not a noticeable difference between the statistics reported by each mapping.

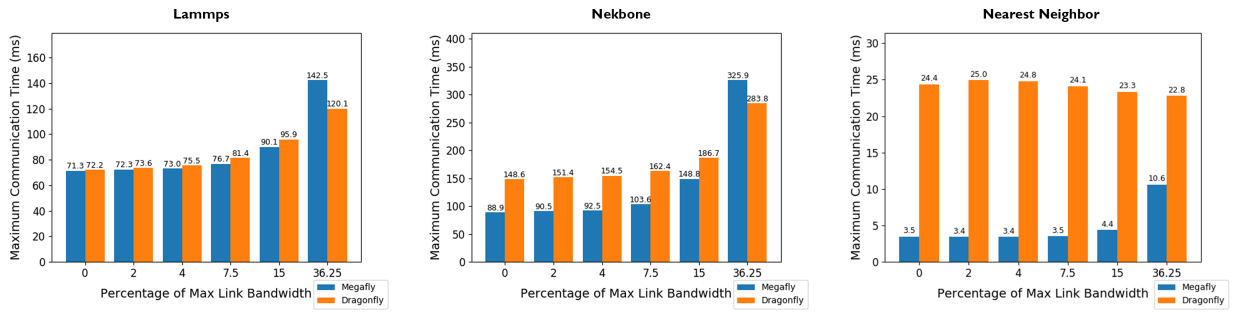
2.5 Quantifying Interference on 1-D Dragonfly and Megafly Networks

In this section, we analyze the communication interference on both 1-D dragonfly and megafly networks following the methodology described in Section 2.4. Each simulation experiment was conducted three times with different geometric job allocation policies, and the performance difference observed between each run was less than 2%. The communication performance is measured by taking the maximum time spent across all ranks participating in the job. Using the maximum time is useful for the case where a few ranks are much slower than the others, which causes the average time to be significantly less than the maximum time. However, the application's communication performance is determined by all participating ranks. The impact of maximum latency on the dragonfly network has been studied in previous work [19]. Before discussing the QoS experiments, we compare the baseline communication performance of megafly and 1-D dragonfly networks and further quantify the performance degradation with QoS disabled. We then incorporate QoS mechanisms in the megafly networks to evaluate the performance with and without such mechanisms enabled.

2.5.1 Uniform Random Background Traffic

Figure 2.4(a) shows the communication time of LAMMPS SWM with varying degrees of background traffic, starting from no background traffic, on both megafly and 1-D dragonfly. LAMMPS uses a mix of point-to-point and collective communication, as shown in Figure 2.3. The performance results in the figure show that megafly performs better than a 1-D dragonfly in most of the background traffic injection rates. For the worst-case background injection rate, 1-D dragonfly outperforms megafly.

Figure 2.4(b) shows the communication time of Nekbone SWM with and without uniform random background traffic on both megafly and 1-D dragonfly networks. Nekbone uses a large number of 8-byte MPI collectives, as shown in Figure 2.3. Additionally, of the studied workloads, Nekbone is the most communication-intensive application: it transmits 4x more data than the LAMMPS or nearest-neighbor SWM workloads do. The performance results in the figure show that megafly performs up to 60% better than a 1-D dragonfly in all except one of the background traffic injection rates. For the worst-case background injection rate, 1-D dragonfly again outperforms megafly.



(a) **(b)** **(c)**

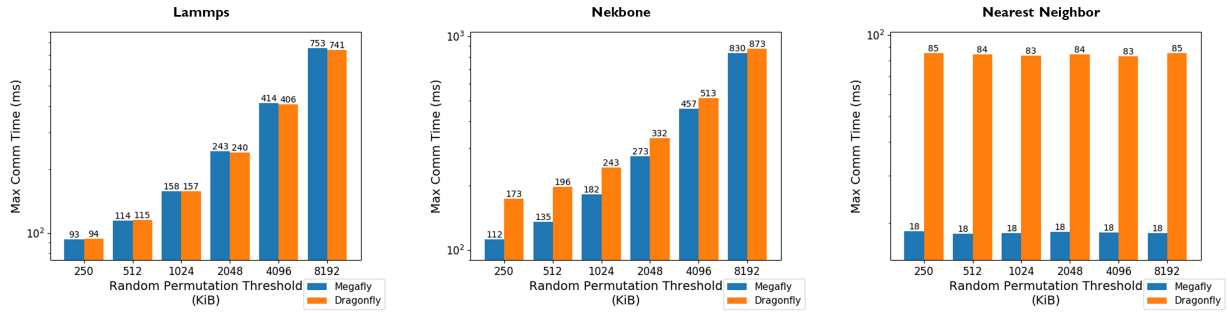
Figure 2.4: Performance of megafly vs. 1-D dragonfly with (a) geometrically allocated 2048/2048 ranks for LAMMPS/uniform random workloads and (b) geometrically allocated 2197/2197 ranks for Nekbone/uniform random workloads, and (c) geometrically allocated 4096/4096 ranks for nearest-neighbor/uniform random workloads. The intensity of the background traffic was scaled at a percentage of the maximum link capacity.

Figure 2.4(c) shows the performance of nearest-neighbor communication on both 1-D dragonfly and megafly networks. Since we are using geometric job mapping that allocates a cluster of network nodes, the nearest-neighbor pattern involves extensive communication between two groups. In this case, megafly consistently outperforms the dragonfly network because of multiple reasons: (i) megafly has four times more links between group pairs than dragonfly, which makes more links available for global connections between the two groups, (ii) megafly has larger group sizes (more nodes available within a group), which increases locality of communication within a group. The locality of communication is beneficial for nearest neighbor traffic, and (iii) Dragonfly has a single minimal path between two routers within a group, whereas megafly has 16 different minimal path options for this route, which reduces intra-group congestion. Since nearest-neighbor communication exchanges are based exclusively on point-to-point operations between two groups, it is less impacted by the background traffic.

2.5.2 Random Permutation Background Traffic

While uniform random traffic with large message sizes can cause dynamic hotspots in the network, we also considered random permutation traffic to introduce more persistent network interference. Similar to uniform random, the random permutation background traffic pattern sends packets randomly to other nodes in the network. We use a rotating random

permutation pattern that will send continually to the same randomly selected destination (on a per-node basis) until a certain number of bytes have been transmitted before choosing a new random destination. Figures 2.5 shows the performance of megafly and dragonfly networks. The foreground workloads see a slowdown in communication time as the number of bytes exchanged in the background traffic is increased. Since nearest neighbor traffic involves point-to-point operations (mostly to the neighboring group), it is not impacted by the rotating random background traffic. While the performance of the LAMMPS workload is comparable on both networks, the Nekbone workload is less perturbed on a megafly network than a 1-D dragonfly. Our conjecture is that the better performance of megafly can be attributed to the additional path diversity of its minimal routes. The results demonstrate that there is nearly a linear slowdown in the performance of the foreground job as the number of bytes exchanged in the background traffic increases.



(a) **(b)** **(c)**

Figure 2.5: Performance of megafly vs. 1-D dragonfly with (a) geometrically allocated 2048/2048 ranks for LAMMPS/random permutation workloads (b) geometrically allocated 2197/2197 ranks for Nekbone/random permutation and (c) geometrically allocated 2197/2197 ranks for nearest-neighbor/Random Permutation workloads. The amount of data exchanged between two nodes in a rotating random permutation was scaled from 250 KiB to 8 MiB.

2.5.3 Multiple Applications in Parallel

In the third case, as a specific instance of representative HPC scenarios, we run the three workloads (LAMMPS, Nekbone, and nearest neighbor) in parallel without any synthetic communication traffic. This scenario clearly mimics a common system state, with multiple jobs completing for shared resources. Figure 2.6 shows the communication time of the three

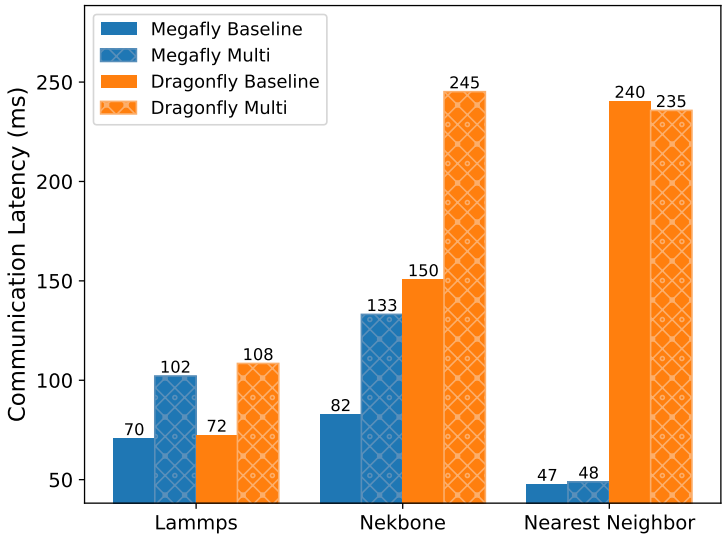


Figure 2.6: Communication time of LAMMPS (2,048 ranks), Nekbone (2,197 ranks), and nearest neighbor (2,048 ranks) when running simultaneously on 1-D dragonfly and megafly networks. Baseline indicates the application runs in isolation.

applications when running in parallel and in isolation on both dragonfly and megafly networks. We can see that with both LAMMPS and Nekbone, the applications are much less perturbed on a megafly network than on a 1-D dragonfly network.

Our analysis shows that in most cases, the megafly network is less prone to perturbation than is a dragonfly network. On both networks, however, HPC applications see a significant slowdown in communication ranging up to 700% in the presence of heavy background communication traffic.

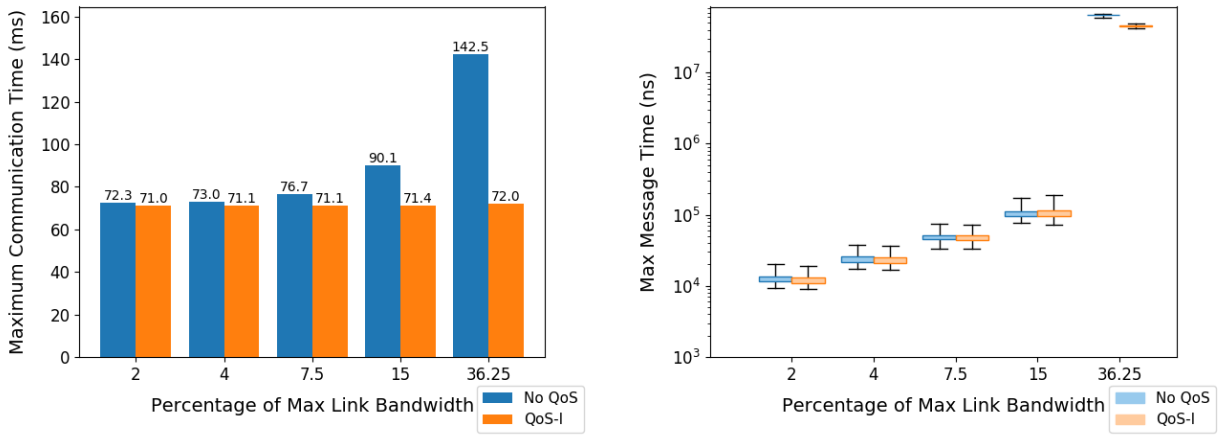
2.6 Evaluating Quality of Service on Megafly Networks

Enabling quality of service on HPC networks requires that each traffic class have its own set of virtual channels. Megafly networks require a fewer number of virtual channels for deadlock prevention. When a fixed, limited number of VCs are available in the switch hardware, megafly needs half as many VCs as a dragonfly and has the opportunity to use the extra VCs for QoS. The mechanism for quality of service was introduced in Section 2.2. In this section, we perform experiments to analyze the impact of QoS on traffic interference and application slowdown that we saw in Section 2.5. We explore two configurations through which QoS can be introduced in megafly networks. Since there can be a large number of permutations

for bandwidth caps, we performed a sensitivity analysis by sweeping different bandwidth values and picked the values that were most effective (for example, a 30% bandwidth cap was effective for multiple applications in Section 2.6.3 and a 10% cap for collective priority in Section 2.6.2). Discussion of the validation of the QoS mechanism is given in [43]. The static window over which the bandwidth statistics were monitored was kept to 5 ms throughout these experiments.

2.6.1 QoS Mechanism I: Prioritizing Entire Applications

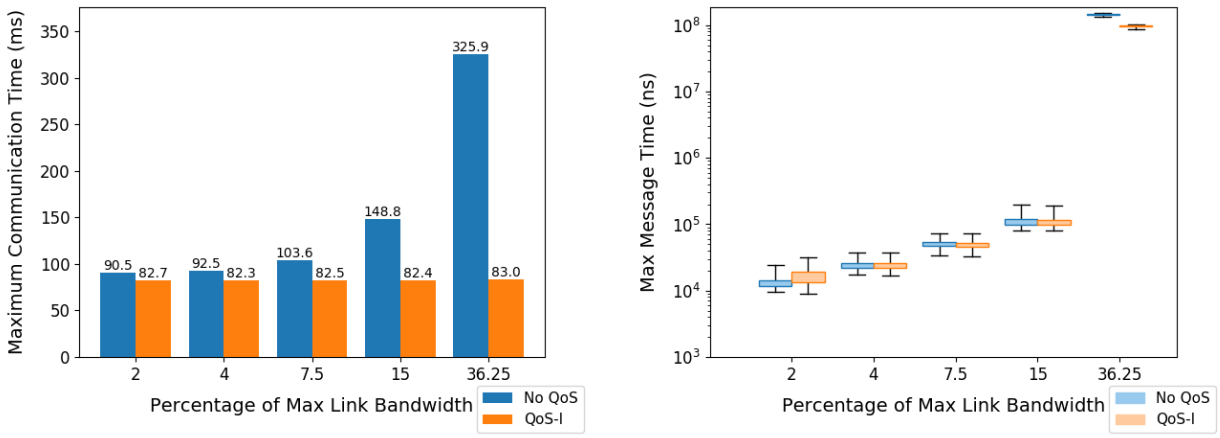
With our first QoS mechanism, a higher priority and high bandwidth are assigned to the entire application (or set of applications) so that they face minimal slowdown relative to other traffic. We use uniform random background traffic and both LAMMPS and Nekbone foreground workloads that exhibited slowdown on megafly networks in Section 2.5 (Nearest neighbor was not getting perturbed). To understand the impact on background traffic, we measure the performance of both foreground workload and background traffic. The background traffic performance is measured by the maximum time to complete a message (all messages have the same size in the synthetic workload).



(a) **(b)**

Figure 2.7: QoS Mechanism I (Application Priority): Performance of LAMMPS (a) and background traffic (b) on megafly network with QoS enabled and disabled. The entire LAMMPS application is given a high priority and high bandwidth (70%).

The benefit of using this QoS approach is that if the foreground application is not utilizing the full bandwidth allocated to it, then the background workload can consume the



(a) **(b)**

Figure 2.8: QoS Mechanism I (Application Priority): Performance of Nekbone (a) and background traffic (b) on megafly network with QoS enabled and disabled. The entire Nekbone application is given a high priority and high bandwidth (70%).

unutilized bandwidth. Figure 2.7 compares the performance of LAMMPS workload with and without QoS enabled on a megafly network. It also shows the slowdown due to background communication traffic. The LAMMPS workload is not as communication-intensive because it involves point-to-point messages along with a small number of MPI_AllReduce messages. Therefore, the perturbation to background traffic is not significant. Because of the high priority given to LAMMPS, it does not see any slowdown even though it is running in parallel with intense background traffic. Additionally, while we observe a significant speedup with LAMMPS, the background traffic observes only a small degree of slowdown as compared with the no-QoS case.

Nekbone SWM is a communication-intensive workload that transmits 4x more data than does LAMMPS SWM, and a majority of the communication involves collectives. Figure 2.8 shows the performance of the Nekbone SWM when QoS is enabled. Once again, we see Nekbone having minimal slowdown when QoS is enabled while causing minimal slowdown to background communication traffic. The primary reason for the improved performance is that both Nekbone and LAMMPS are given high priority and high bandwidth, yet they do not consume all the bandwidth assigned to them. Therefore, the background traffic is able to get the required bandwidth that it needs while observing little slowdown. Both these results demonstrate that traffic differentiation with bandwidth shaping and prioritization

can mitigate (or eliminate) communication interference to HPC workloads while causing minimal slowdown to the background traffic. Assigning a high priority to an application can eliminate the perturbation to that application while experiencing a reasonable slowdown in the remaining network traffic.

2.6.2 QoS Mechanism II: Prioritizing Latency-Sensitive Operations

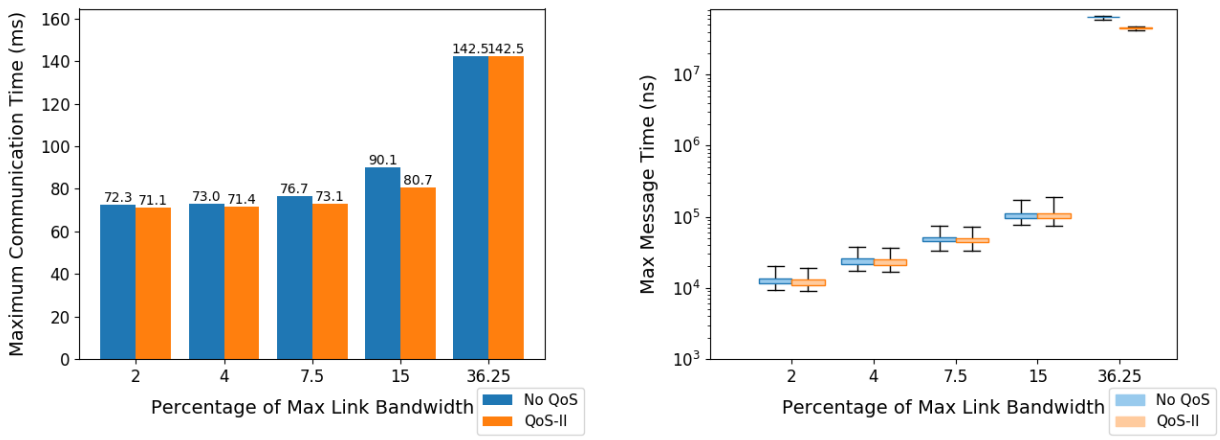
Several HPC applications rely on the performance of MPI collective operations. In a majority of the cases, collectives comprise small messages, and the application performance suffers when heavy background network traffic interferes with the transmission of these messages. An alternative application of QoS is to assign a high priority and guaranteed bandwidth to collective operations.

Figure 2.9 shows the performance of LAMMPS when high priority is given to collectives and compares it with the case where no QoS is enabled. In this case, we are assigning a high priority but a small fraction of bandwidth to collective operations; the point-to-point operations and background traffic are given a lower priority and higher bandwidth cap (90%). We see that although there is some slowdown in foreground traffic when the background traffic becomes intense, the foreground workload is still 10% faster than the case where no QoS is enabled (specifically in the case of 15% background traffic injection). Additionally, LAMMPS uses more point-to-point operations and has fewer collective operations.

Figure 2.10 shows the performance of Nekbone when given high priority and a guaranteed amount of bandwidth to collectives. Nekbone relies heavily on collective operations, whereas LAMMPS uses comparatively fewer collectives. Therefore, we see a significant performance improvement of up to 60% speedup compared with the case where no QoS is enabled. The background communication traffic does not show a slowdown in message communication times. Instead, it shows a slight performance improvement compared to no-QoS options in one case.

2.6.3 Applying QoS Mechanisms to Multiple Application Workloads in Parallel

In this section, we examine both QoS mechanisms in the case where multiple applications are running in parallel, which is a specific instance of a representative HPC system. We compare the QoS-enabled performance with the case where there are multiple applications running without any QoS. For the first QoS mechanism, since Nekbone is more communication-



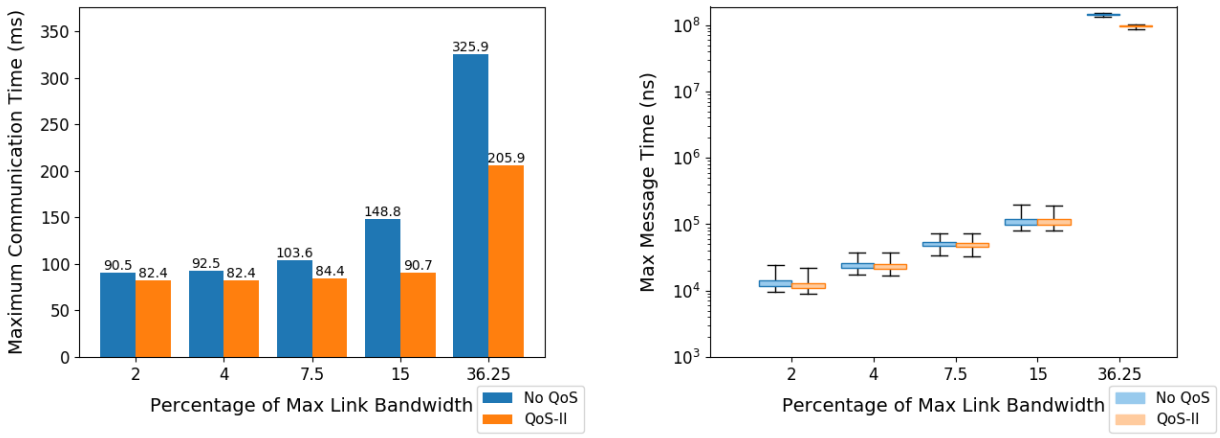
(a) **(b)**

Figure 2.9: QoS Mechanism II (Collective Priority): Performance of LAMMPS (a) and background traffic (b) on megafly network with application-based QoS enabled and 10% bandwidth guaranteed to collectives.

intensive than LAMMPS and nearest neighbor (shown in Figure 2.3), we assign it a separate traffic class with a bandwidth cap of 30% and a high priority. The rest of the bandwidth (70%) is available to both LAMMPS and nearest neighbor. For the second QoS mechanism, we assign a higher priority to all collective communication in both LAMMPS and Nekbone and then see the impact on application performance.

Figure 2.11 shows the performance of multiple applications running in parallel with and without QoS enabled. In short, both schemes are beneficial and lead to reduced communication time. With the QoS Mechanism I, we give a high priority and guaranteed bandwidth to Nekbone (30%). Nekbone is communication-intensive. With a high priority and 30% of the bandwidth cap, it does not get any slowdown due to background communication traffic. In contrast, LAMMPS and nearest neighbor have a lower priority, and they still see a performance improvement compared with the case where there was no QoS enabled. Adding bandwidth caps on Nekbone (which is a bandwidth-intensive application) helps improve the performance of LAMMPS and nearest neighbor as well.

With the QoS Mechanism II, where collective communication is given priority, both LAMMPS and Nekbone benefit by seeing a 10% and 20% speedup, respectively, compared with the case where QoS is not enabled. One interesting observation is the performance of the nearest-neighbor workload, which is much faster with QoS enabled than when the



(a) **(b)**

Figure 2.10: QoS Mechanism II (Collective Priority): Performance of Nekbone (a) and background traffic (b) on megafly network with application-based QoS enabled and 10% bandwidth guaranteed to collectives.

workload is running in isolation. Looking at the adaptive routing statistics, we see that the nearest-neighbor workload when running in isolation takes the maximum number of minimal routes because of the bias toward minimal routes. With QoS enabled, nearest-neighbor traffic has a lower priority with QoS mechanisms enabled, which causes it to take more nonminimal routes but is largely unaffected. Thus the workload sees improved performance. A similar phenomenon is observed with Nekbone when it is running with QoS Mechanism I.

These experiments demonstrate the effectiveness of applying QoS to reduce or eliminate communication interference. With both mechanisms, Nekbone, being more bandwidth-intensive, sees a 20% to 350% speedup in communication time compared with the case where QoS is not enabled. LAMMPS sees a 10% to 200% improvement in communication time compared with the case where QoS is not enabled.

2.7 Discussion & Conclusion

With HPC applications showing performance variation on recent hierarchical interconnects, we analyze communication interference for both megafly and dragonfly networks. We extend the CODES parallel simulation framework to replay the communication workloads of LAMMPS, Nekbone, and nearest neighbor using the concept of Scalable Workload Models (SWM). We introduce moderate to intense background communication traffic during the

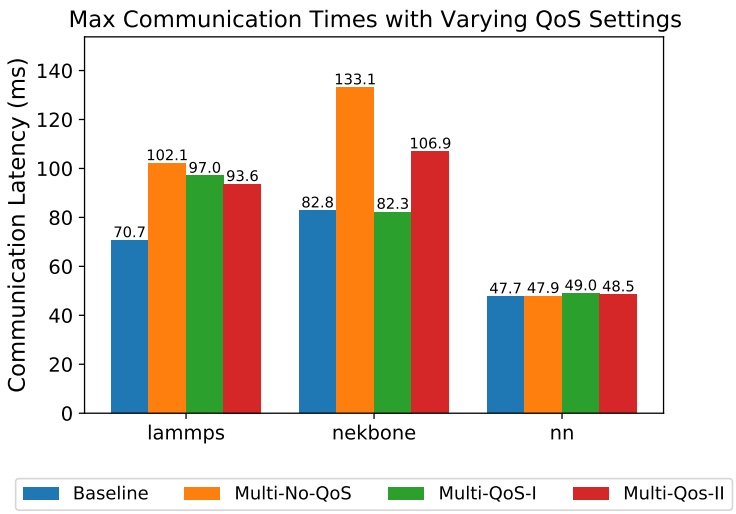


Figure 2.11: Communication time of Nekbone, LAMMPS, and nearest-neighbor workloads when running in parallel. Both mechanisms of application-based QoS were enabled. The comparison is made with (i) the worst case when no QoS is enabled (Multi No QoS) and (ii) the best case when the workload is running in isolation with no interference (baseline).

execution of these communication workloads and compare the slowdown on the megafly network with that on a 1-D dragonfly network. We demonstrate that performance variability is experienced in both topologies while also observing that in a majority of experiments, the performance implication is less severe for megafly.

To further mitigate the variability, we introduce traffic differentiation and quality of service mechanisms in megafly networks, as megafly makes QoS implementation feasible by requiring fewer VCs for deadlock avoidance. We explore two different QoS mechanisms for HPC workloads (i) prioritizing and bandwidth capping entire HPC applications (ii) prioritizing and guaranteeing bandwidth to latency-sensitive collective operations with small messages. With the first mechanism, performance results show that when a high priority and a bandwidth cap is given to entire HPC applications, it can eliminate performance variability while the rest of the background traffic also sees minimal impact. For the second mechanism, we show that when a small fraction of bandwidth is guaranteed to latency-sensitive operations like the MPI collectives, it can mitigate the performance variability by 10% to 60% depending upon the intensity of collective communication in the application.

While this work is aimed to provide a proof of concept that QoS is effective in mitigating

communication interference for realistic HPC workloads, there are a number of avenues that need to be further explored. First, real HPC systems have tens to hundreds of jobs running. Giving a high priority to more than one HPC application (as shown in QoS mechanism I) can introduce interference within the traffic class, which can slowdown high priority applications. Secondly, a statically allocated time window for bandwidth monitoring can lead to bandwidth imbalance with bursty and irregular workloads. Having a dynamic or sliding bandwidth monitoring window may be more effective for such cases. Finally, one would need to explore how to expose the traffic classes to the MPI interfaces and the scheduler.

Future work could also explore different improvements to the QoS implementation, including an optimization that removes the static time window constraint in favor of a more dynamic approach, an expansion to an arbitrary number of traffic classes, and how quality of service handles problematic traffic patterns such as the many-to-one.

CHAPTER 3
FIT FLY: A CASE STUDY ON INTERCONNECT INNOVATION THROUGH PARALLEL SIMULATION

To meet the demand for exascale-level performance from high-performance computing (HPC) interconnects, many system architects are turning to simulation results for accurate and reliable predictions of the performance of prospective technologies. Testing full-scale networks with a variety of benchmarking tools, including synthetic workloads and application traces, can give crucial insight into what ideas are most promising without needing to physically construct a test network.

While flexible, however, this approach is extremely compute time intensive. We address this time complexity challenge through the use of large-scale, optimistic parallel simulation that ultimately leads to faster HPC network architecture innovations. In this paper we demonstrate this innovation capability through a real-world network design case study. Specifically, we have simulated and compared four extreme-scale interconnects: Dragonfly, Megafly, Slim Fly, and a new dual-rail-dual-plane variation of the Slim Fly network topology.

We present this new variant of Slim Fly, dubbed Fit Fly, to show how interconnect innovation and evaluation—beyond what is possible through analytic methods—can be achieved through parallel simulation. We validate and compare the model with various network designs using the CODES interconnect simulation framework. By running large-scale simulations in a parallel environment, we are able to quickly generate reliable performance results that can help network designers break ground on the next generation of high-performance network designs.

3.1 Introduction

When designing a new high-performance computing (HPC) system, the choice of the underlying interconnection network is not to be taken lightly. Performance and cost must be balanced appropriately to prevent large bottlenecks. Moreover, different configurations of the same topology can have varying levels of performance depending on the types of

This chapter previously appeared as: N. McGlohon, N. Wolfe, M. Mubarak, and C. D. Carothers, "Fit Fly: a case study on interconnect innovation through parallel simulation," in *Proc. ACM SIGSIM. Conf. Princ. Adv. Discrete Simul.*, 2019, pp. 137–148.

communication featured in a given workload. Simple synthetic workloads cannot reproduce all of the complexities found in real-world applications [45]. Thus, HPC system integrators must rely on exhaustive and fine-grained simulations, testing various configurations, topologies, and workloads to pick the right interconnect for their system. As network features such as routing algorithms, congestion-abatement schemes, and fault tolerance mechanisms get more complicated, predicting full-scale, real-world performance of prospective system designs becomes even more challenging.

As mentioned in Chapter 1.4, we leverage the CODES (Co-Design of Exascale Storage) storage and network simulation toolkit to develop, test, and evaluate exascale topologies and technologies. Previous works have implemented and tested the performance of routing schemes such as On-the-Fly Adaptive Routing (OFAR) [46] or Universal Globally-Adaptive Load-balanced (UGAL) routing [47] on networks featured in the CODES toolkit [21], [48]–[50]. We show that this framework can be used to evaluate not only existing networks but also newly proposed networks that have not yet been built in a quick and efficient time frame.

We propose Fit Fly, a variant of the Slim Fly network topology [16], and compare it with state-of-the-art hierarchical networks such as Dragonfly [13] and Megafly [15]. Fit Fly has the same basic topology and construction as Slim Fly but features a dual-rail-dual-plane configuration. Fit Fly is a product of "what-if?" questions: What if we could (1) double the network's overall bandwidth, (2) further reduce the average number of hops between two terminal nodes in the network, and (3) double the number of routers to reduce the possibility of congestion?

Because of its second, independent plane of routers, Fit Fly does incur twice the router cost, but with this cost comes twice the overall bandwidth. Because of the way Slim Fly networks are constructed, they already have high path diversity and low diameter. If we add a second plane of routers "on top" of the first but with the terminal nodes mapped in a mirrored fashion, we can further increase the path diversity and reduce the average number of hops between two nodes. Moreover, with the second injection rail and increased overall bandwidth, packets can be injected into the network at a rate much higher than a single-rail-single-plane Slim Fly network. Additionally, we found that even at a reduced link bandwidth, a second plane of routers greatly reduced the impact of interference traffic on application performance.

The main contributions in this chapter are as follows:

1. A variant of the Slim Fly topology and ROSS+CODES network model that enables multiple independent router planes with multiple injection rails at each network terminal.

2. An example methodology for how hypothetical questions can be quickly answered through rapid development and experimentation with the CODES framework. This is demonstrated through a series of experiments on four state-of-the-art network designs of approximately 3,000 nodes.

3. A corroboration of the conclusions found in [16], [51] relating low-diameter networks to increased performance as well as the benefits of multiple router planes in combating network congestion.

4. Evaluation of Slim Fly and Fit Fly with comparable overall bandwidth showing that the increased performance of Fit Fly over Slim Fly can be maintained even with reduced-cost hardware.

3.2 Fit Fly Network Model

The Fit Fly model is an enhanced version of the Slim Fly model previously added to the network set of CODES [50]. The topology of a Fit Fly network is thus similar to that of Slim Fly, with a lot of overlapping details. Unless otherwise specified, any aspect that applies to Slim Fly also applies to Fit Fly.

Table 3.1: Description of symbols used to define Slim Fly and Fit Fly networks.

h	Nodes connected per router
P	Number of independent router planes
N_{r_p}	Total routers per plane
N_r	Total routers in network ($N_r = N_{r_p} \cdot P$)
N_h	Total nodes in the network ($N_h = N_{r_p} \cdot h$)
k'	Router network radix
k	Router radix ($k = k' + h$)
q	Prime power

The most critical difference between Slim Fly and Fit Fly is that the latter has at least two independent router planes and rails, whereas the default Slim Fly model only has one. Each plane contains N_{r_p} routers, which are connected only to other routers in the same plane.

While Fit Fly has multiple router planes, the total number of nodes in the network remains the same. Nodes, or terminals, within the network are mapped to routers in each plane by using a scheme detailed in Section 3.2.2.2.

3.2.1 Slim Fly Background

In a work that introduced the Slim Fly topology, Besta et al. [16] argued that lowering the network diameter could be beneficial in several ways. First, the lowered network diameter could translate to reduced latency because, on average, fewer hops would be traversed by packets. As a result, packets would be less likely to interfere with each other. Second, fewer routers are necessary to create a connected network, so there is a reduced cost in terms of up-front construction as well as maintaining energy costs. Besta et al. showed that a low-diameter network could be designed without sacrificing high overall bandwidth and while simultaneously reducing costs.

The main objective in the design of the Slim Fly topology was to maximize the number of endpoints N_h with a given network diameter D and vertex (router) radix k. (For notation, see Table 1.) Besta et al. [16] drew parallels between this objective and that of the *degree-diameter problem* [52]. They thus decided to use graphs related to that problem for the backbone of Slim Fly. In order to maximize the number of endpoints but also guarantee the low diameter property, the network is constructed based on a class of diameter-2 graphs commonly referred to as MMS graphs. MMS graphs of diameter-2 are close to the optimum size noted in the degree-diameter problem as they approach what is referred to as the Moore bound [53].

3.2.2 Slim Fly Topology

Slim Fly network routers are divided into local groups, each with a certain degree of local connectivity. The set of all local groups is divided between two subgraphs that, for ease of reference, we will call the α and β subgraphs.

Each router in a local group also has a certain degree of global connectivity. We define a global connection as any router-router connection that spans between two local groups. Similarly, local connections are defined as any router-router connections that fall within a local group. *In Slim Fly, there are no connections between groups within the same subgraph.*

The result of these specifications is a bipartite graph with global connections spanning

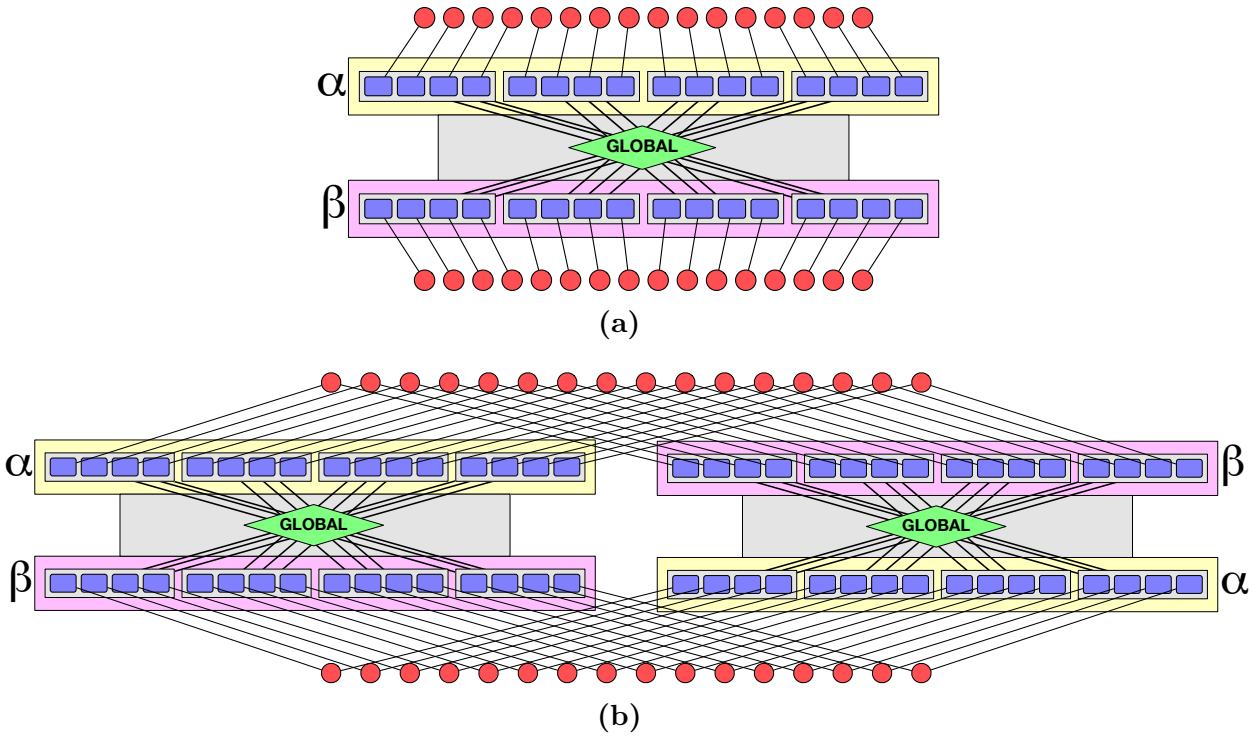


Figure 3.1: General layout of the Slim Fly and Fit Fly topologies. Networks shown are simplified for ease of understanding; normal networks will have far more routers and terminal nodes per router. (a) shows an example of the Slim Fly topology. Each terminal has a single connection to a router (single-rail) in either subgraph α or subgraph β. (b) shows an example of the Fit Fly topology. Each terminal has two connections (dual-rail) to routers, one connection to a router in each of two planes of independent routers (dual-plane). One distinction of Fit Fly is that the second plane of routers is identical to the first but mirrored, increasing the total path diversity further.

the two subgraphs and several local groups within each subgraph with local connections between routers in each group.

Each router also has a certain number of terminal nodes connected to it. Figure 3.1a shows an example Slim Fly layout. Labeled are the α and β subgraphs, each with four router groups of four routers. Each router in this example has a single terminal host attached. The exact connections from router to router, both within local groups and globally, are not shown in detail because the construction is not trivial. Details of this construction are given in Section 3.2.2.1.

3.2.2.1 *Constructing a Slim Fly-Class Network*

Given the correct parameters for generation of the topology, the simulation model will calculate the nontrivial link structure and connect the router LPs to their corresponding neighbors. Finding the correct parameters to input is itself nontrivial, however. To determine the correct input parameters to define the network, we follow the construction-simplified methodology for Diameter-2 MMS graphs laid out in greater detail in [16]. According to those authors (with slight modification for multiple planes), this process comprises five steps.

1. Choose the number of planes p that will exist in the network. Standard Slim Fly has a single plane of routers, whereas Fit Fly has at least two.

2. Find a prime power $q = 4\omega + \delta$, where $\delta \in \{-1, 0, 1\}$, such that $N_{r_p} = 2q^2$ is satisfied for the desired number of routers per plane.

3. Construct a Galois field of order q. Let $\mathbb{F}_q$ be such Galois field. Also, find the primitive element ξ that generates it. ξ is an element that *generates* all other elements of the set. More formally, all nonzero elements of $\mathbb{F}_q$ can be written as ξ^i, where $i \in \mathbb{N}$.

4. Utilizing ξ, construct sets X and X', known as *generator sets*. These generator sets specifically will be used to determine router-router connections within a single network plane using Equations 1–3.

5. Connect terminals sequentially to routers, reversing the order for each consecutive plane.

We can assign each router in the network with four coordinates (s, x, y, p), where $s \in \{\alpha, \beta\}$ indicates which of the two subgraphs the router is in. $x \in \{0, \ldots, q-1\}$ and $y \in \{0, \ldots, q-1\}$ represent the local group that the router resides in and its position within the group, respectively. The fourth coordinate, $p \in \{0, \ldots, P\}$ represents the network plane. Since each plane's router connections are independent and identically computed, this process is repeated for each plane p.

$$\text{router}(\alpha, x, y, p) \text{ connected to } (\alpha, x, y', p) \text{ iff } y - y' \in X \tag{3.1}$$

$$\text{router}(\beta, m, c, p) \text{ connected to } (\beta, m, c', p) \text{ iff } c - c' \in X' \tag{3.2}$$

$$\text{router}(\alpha, x, y, p) \text{ connected to } (\beta, m, c, p) \text{ iff } y = mx + c \tag{3.3}$$

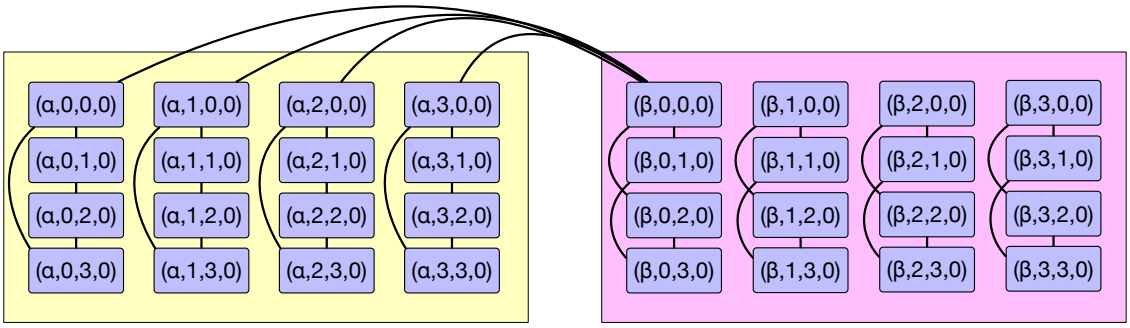


Figure 3.2: Example visualization showing the three distinct ways connections could be determined. Many global connections are not shown for simplicity. Note that this is for a single plane, denoted by the fourth coordinate. The same process is used for all planes, and there are no connections between routers on different planes.

We utilize three equations presented in [16]. Equation 1 is used to compute the intragroup (local group) connections of subgraph α. Similarly, Equation 2 computes the intragroup connections within subgraph β. Equation 3 determines the connections that span between the two subgraphs. A toy example of how the connections can be visualized is presented in Figure 3.2.

3.2.2.2 *Fit Fly: Additional Rails and Planes*

Unless otherwise noted, Fit Fly refers to a Slim Fly network with two independent planes of routers as well as two injection rails on each terminal for introducing and receiving traffic to/from the network. This makes Fit Fly a dual-rail-dual-plane network by default, but it can also be generalized to refer to a multi-rail-multi-plane Slim Fly network of arbitrary order.

As mentioned in Section 3.2.2.1, Equations 1–3 are used to determine the connectivity within a single network plane. Since Fit Fly has at least two planes, we must perform the same calculations for each plane. We emphasize that there are no connections between any two routers on separate planes.

What joins the planes together to form a single cohesive network are the terminals. The multiple planes share the same set of terminal nodes. Figure 3.1 shows a general layout of both Slim Fly and Fit Fly. In Figure 3.1a, each terminal has a single connection to a router, following the standard Slim Fly design. Figure 3.1b shows each terminal having a connection to a single router on each plane.

While the router connections within each plane are identical, the mapping of terminals to routers in each plane is not. When connecting terminals to a Fit Fly network, the terminals are connected via two schemes depending on the plane being linked to. If we number the planes with the ID p in the range [0,P), the mapping is as follows. In even-numbered planes, terminals are mapped with the default Slim Fly scheme; this means that terminal node 0 is connected to router (α, 0, 0, p) and so on. In odd-numbered planes, however, terminals are mapped in reverse; terminal 0 is connected to router (β, $q-1$, $q-1$, p) and so on in those planes. This scheme increases the overall path diversity and reduces the average hop count.

3.2.3 Routing and Planar Selection

Currently, three algorithms for packet routing within a plane are implemented in the Slim Fly and, by extension, Fit Fly models: minimal, nonminimal, and adaptive. These routing algorithms remain unchanged from previous work [50].

While Fit Fly utilizes the same routing algorithms as does Slim Fly on a per plane basis, an additional scheme must be in place in order to determine which rail, and consequently which plane, a packet will be passed through at the originating terminal. Three different schemes have been implemented for this selection: `PATH`, `CONGESTION`, and `RANDOM`. The best choice of planar selection algorithm depends on what type of experiments are to be performed.

3.2.3.1 *`PATH` Planar Selection*

When the network simulation is configured to utilize the `PATH` selection algorithm, the originating terminal evaluates the minimal path length to the destination terminal along each rail. The shortest path is then chosen. In the event of a tie for the shortest path, the `CONGESTION` algorithm is used.

3.2.3.2 *`CONGESTION` Planar Selection*

The `CONGESTION` scheme evaluates the number of packets set for injection along each rail at the originating terminal. Whichever rail has the fewest number of packets awaiting injection will be chosen. This method is intended to increase the load-balancing properties of the network and prevent performance degradation from a single rail becoming overly congested.

3.2.3.3 *RANDOM Planar Selection*

A more trivial way to balance load across the various planes is to allow for the originating terminals to select the rail of injection uniformly at random.

3.3 Validation

The Slim Fly CODES network model has previously been validated against an independently developed Slim Fly simulation for minimal, nonminimal, and adaptive routing [50], [54]. Fit Fly utilizes the same base model as Slim Fly with minimal changes to enable links to the additional rails/planes. Because no real-world Fit Fly model or other simulation exists, validation can be performed only in comparison with the Slim Fly model itself.

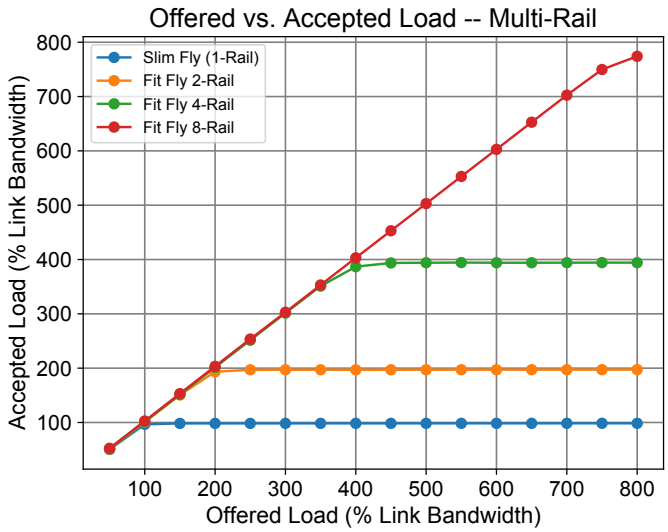


Figure 3.3: Slim Fly, dual-rail, quad-rail, and octo-rail Fit Fly networks were offered varying loads by sweeping the mean amount of time between when packets of fixed size were injected into the networks. The number of router planes in each network strictly equals the number of rails. Here the load is shown as a percentage of the link bandwidth (12.5 GiB/s).

Since Fit Fly's main feature is its increased overall bandwidth, we have performed a set of offered versus accepted load experiments. These experiments show that the additional rails and planes do, in fact, yield a greater accepted load amount that is proportional to the number of rails in the network. While Fit Fly generally refers to dual-rail-dual-plane Slim Fly, we have expanded the model to allow for arbitrary numbers of rails/planes to show flexibility.

Table 3.2: Configurations of Slim Fly, Fit Fly, Dragonfly, and Megafly networks used for performance comparison.

	Slim Fly	Fit Fly	Dragonfly	Megafly
Router Radix	28	28	36	36
Planes	1	2	1	1
Rails	1	2	1	1
Groups	26	52	19	10
Node Count	3042	3042	3078	3240
Router Count	338	676	342	360
Global Connections	4732	9464	3078	3240
Nodes/Group/Rail	117	117	162	324
Global Conn. / Group	169	169	162	324
Link Bandwidth	12.5 GiB/s	12.5 GiB/s	12.5 GiB/s	12.5 GiB/s
Nodes per Router	9	9	9	18 (Leaves only)
Routing Algorithm	Adaptive (UGAL)	Adaptive (UGAL)	Adaptive (PAR) [44]	Adaptive (PAR)
Planar Selection	n/a	`CONGESTION`	n/a	n/a

Figure 3.3 shows the results of these runs. The accepted load was measured by counting the number of bytes received by terminals and dividing by the duration that the count was measured. Additional rails give the networks greater capacity and overall bandwidth and are therefore able to accept greater amounts of offered load.

3.4 Experiments

In this section, we describe the various experiments that we have performed to compare the Fit Fly network model with others of similar size in terms of the number of both terminals and routers.

For all experiments in this work, we allocate terminals for both a primary workload and a secondary, background, workload with the exception of zero interference (baseline) runs. The same random allocation map was utilized for each network.

3.4.1 Workloads

The networks in this chapter are evaluated with two separate HPC network traces provided by the Design Forward Program [31].

We used traces from the following applications: AMG and MultiGrid. Note that each of these traces are sourced from applications with multiple ranks per host but are replayed on our network models with a 1:1 rank/host ratio to spread out communication across the network, forcing hosts from all parts of the network to participate in the primary workload.

3.4.1.1 *Algebraic MultiGrid Solver (AMG) Workload*

AMG is an algebraic Multigrid solver mini-app for unstructured mesh physics packages. The specific trace used in our experiments has 1,728 MPI ranks. We map each rank to its own terminal node in the given network. The application from which this trace was created had a runtime of 2.2 seconds, including computation time, with 1,728 MPI ranks on 72 hosts and 52% of the time spent on communication. [31]

3.4.1.2 *MultiGrid (MG) Workload*

MultiGrid is a geometric multigrid cycle mini-app from the adaptive mesh refinement application framework BoxLib [55]. The specific trace used in our experiments has 1,000 MPI ranks, each of which, similarly to our AMG workload, is mapped to its own terminal node in the simulated network. The application from which this trace was created had a runtime of 1.4 seconds, including computation time, with 1,000 MPI ranks on 42 hosts. The application spent 3.7% of the time on communication [31].

3.4.1.3 *Synthetic Workload*

In addition to the primary workloads (AMG and MG traces), we use a synthetic uniform random workload to simulate various levels of induced traffic load on the network. With a mean interval of 100μs, we vary the payload size to alter the intensity of interference.

3.4.2 Evaluation Metrics

The three primary metrics that we analyze are the maximum communication time of the primary workload, the average packet latency of all packets in the network, and the average number of hops traversed by all packets in the network. The maximum communication time is the total amount of time spent by any one rank of the primary workload from the first MPI message it sends to the final. The average packet latency is important for qualifying how long packets are spending in transit and in router buffers. This metric is correlated with the communication time but is application agnostic. The total number of hops traversed can give insight into how much deviation packets experience as a result of the adaptive routing algorithm due to congestion.

3.4.3 Other Evaluated Networks

We evaluate four state-of-the-art networks: Slim Fly, Fit Fly, Dragonfly, and Megafly. Since in Section 3.2 we already discussed the specifics of how Slim Fly and Fit Fly topologies are designed, this section covers some background of the latter two networks.

3.4.3.1 *Dragonfly*

The specific type of Dragonfly network that we simulate in this work is a 1D Dragonfly topology introduced in [13]. It features a hierarchical design of groups of routers. Routers within groups are locally connected in an all-to-all pattern. In addition to local and terminal node connectivity, routers have some degree of global connectivity (links to routers in other groups). *Every group must have at least one global connection to every other group in the network.*

This type of Dragonfly network is not to be confused with the 2D-Dragonfly topology featured in the Cray XC systems known as Cray Cascade [14]. While there are similarities, the local groups of the 2D-Dragonfly feature a grid-type connectivity instead of all-to-all.

3.4.3.2 *Megafly*

The network type called Megafly, also known as Dragonfly+, is a derivative of the 1D Dragonfly network introduced in [15]. Whereas Dragonfly has all-to-all local connectivity, Megafly features a two-level fat tree for each local group. The result is that within each group of routers is a fully-connected bipartite graph. On one half are routers, called *leaf routers*, that have terminal node connections and local connections. On the other half of the bipartite graph are routers, called *spine routers*, that have global connections to other groups and local connections. Spine routers have no terminal connections, and leaf routers have no global connections. Similar to Dragonfly, every group must have at least one global connection to every other group in the network.

3.4.4 Cross-Network Comparison

In this section, we show a direct comparison of the Fit Fly model not only to Slim Fly but also to a couple of dragonfly-class networks. The intention is not to argue that one network is definitively better than another but instead to show how flexible the CODES framework is for quickly testing multiple types of networks with varying workloads and

interference intensities. We feel that these networks also give some added context for the power of Slim Fly-class networks.

Table 3.2 gives the parameters of the four networks tested in this section. Because of the difference in the guidelines or restrictions for generating these networks, a true comparison with all aspects of the networks being the same is difficult. As a result, our Slim Fly network has many more global connections than either Dragonfly or Megafly has. Fit Fly, because of its second router plane, has even more.

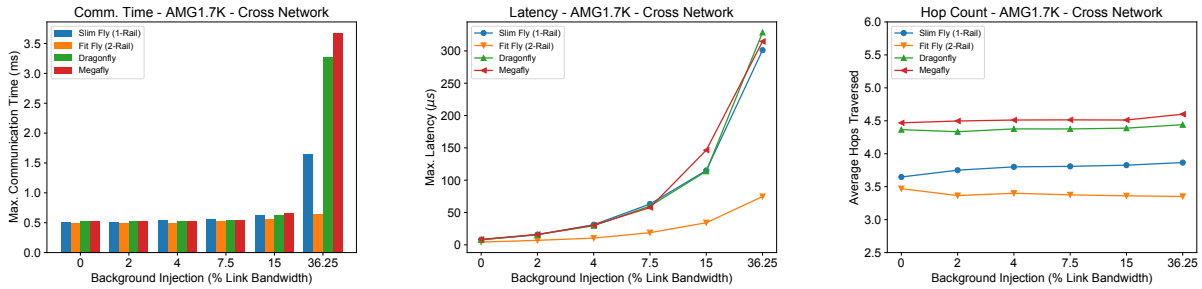
The link bandwidth that we have chosen for our network configurations is to match the Mellanox InfiniBand EDR specifications. However, we have used 12.5 GiB/s instead of 12.5 GB/s for configuring our networks in order to match previous experiments. As a result, our bandwidth configurations slightly exceed their respective Mellanox InfiniBand counterpart by approximately 7% but should negligibly affect overall observed trends and final results.

Figure 3.4 shows the results of the experiments performed on the AMG1728 workload.

In Figure 3.4a, we note that the networks are closely matched at low levels of background interference but start to exhibit some performance loss at higher levels of traffic injection. We can see that Fit Fly well outperforms each network, which is expected by looking at the statistics from Table 3.2. We reason that Fit Fly performed so well because of its increased overall bandwidth: it has twice the routers and thus twice the number of links for packets to be transmitted across and, therefore, less opportunity to encounter interfering congestion.

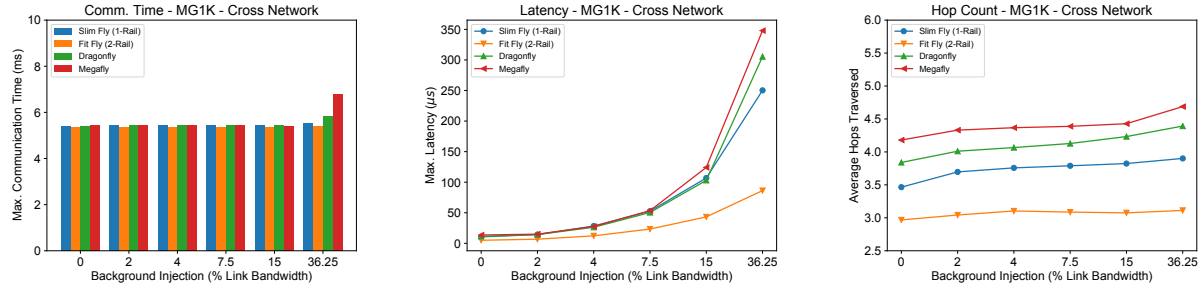
Comparing the average number of hops traversed by packets in the network can also give insight into how well each network handles congestion. Figure 3.4c plots the average number of hops traversed by all packets in the network, not taking into account the different workloads. Slim Fly and Fit Fly show a great advantage over the Dragonfly and Megafly networks in this regard as well because of the low diameter of the Slim Fly-based topologies. Fit Fly again takes the lead, notably because the extra routers give it a distinct advantage for packets to be routed more directly to their destination. With a constant packet injection in a load-balanced network, the likelihood that any two packets will find themselves queued on the same router decreases as the number of routers increases.

Figure 3.5 shows similar results from the MultiGrid1000 trace workload. In Figure 3.5a we observe that each network appears to handle increasing levels of background traffic up to 15% of link bandwidth unperturbed, with Fit Fly narrowly outperforming them all. Fit Fly remains unimpeded even at the highest level of background interference. Figure 3.5c shows



(a) (b) (c)

Figure 3.4: Synthetic interference experiments showing (a) communication time, (b) packet latency, and (c) hop count on the AMG1728 trace workload with 1,000 synthetic background ranks. Each synthetic rank injects packets into the network at a fraction of the total link bandwidth of 12.5 GiB/s (≈InfiniBand EDR). The same workloads and allocations were used across four networks.



(a) (b) (c)

Figure 3.5: Synthetic interference experiments showing (a) communication time, (b) packet latency, and (c) hop count on the MultiGrid1000 trace workload with 1,000 synthetic background ranks. Each synthetic rank injects packets into the network at a fraction of the total link bandwidth of 12.5 GiB/s (≈InfiniBand EDR). The same workloads and allocations were used across four networks.

that Fit Fly did not need to reroute many packets, on average, away from a minimal path.

We note that the convention that CODES follows for counting hops is that the count is incremented when received by a router in the network—not on receipt by a terminal. Thus, a packet that originates at a terminal node, visits two routers, and then is received by the destination terminal will have a path hop count of two.

3.4.5 Equalized Bandwidth Comparisons

The configuration of Fit Fly given in Table 3.2 was chosen to change as few variables as possible from the chosen Slim Fly configuration. Thus, since our Slim Fly network was configured with link bandwidth speeds close to Mellanox InfiniBand (IB) EDR 100 Gb/s specifications, so too was Fit Fly.

This decision, however, gives Fit Fly a distinct advantage. The amount of overall bandwidth, the total amount of bandwidth available to link any two terminals in the network, is far greater in the case of Fit Fly. Additionally, Fit Fly has a higher number of routers than do the other networks by a factor of 2.

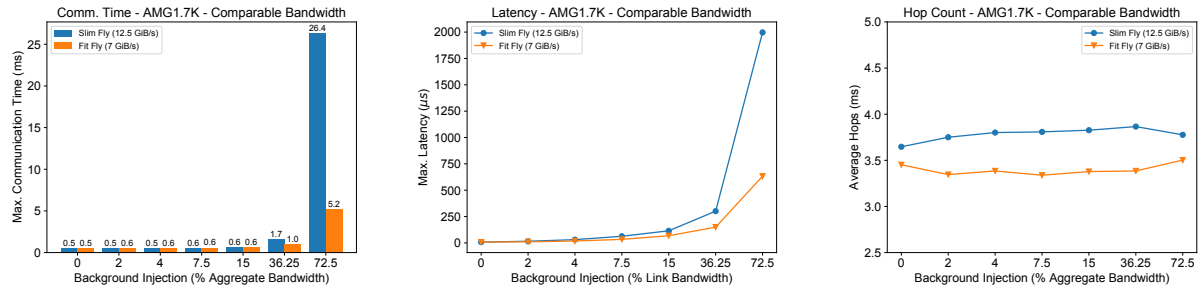
To provide a different perspective, we perform additional experiments where the bandwidths of the two networks are configured to be mutually comparable. In one set, we halve Fit Fly link bandwidths to match Mellanox IB FDR 56 Gb/s link specifications as well as experiments where the bandwidth of Slim Fly is instead doubled to match Mellanox IB HDR 200 Gb/s link specifications. The result is two pairs of Slim Fly and Fit Fly networks where the total overall bandwidth between networks in each pair is of a comparable magnitude.

The first comparison uses the same Slim Fly network described in Table 3.2 but with a modified Fit Fly network where all link bandwidths are nearly halved to match the 56 Gb/s link speed specification. The results of these experiments are shown in Figures 3.6 and 3.7. In these figures, we observe that while the networks are uncongested, they have similar application performance in each trace. When the network experiences higher levels of injection traffic, however, Fit Fly is able to handle larger levels of interference, even with reduced bandwidth.

Similarly, if instead of halving the bandwidth of Fit Fly, we double the bandwidth of Slim Fly to the level of Mellanox IB HDR 200 Gb/s, we observe the same phenomenon. Normalizing for bandwidth shows that bandwidth is only part of the story. With the total amount of bandwidth between the two networks kept at comparable levels, the only true difference between the two networks is the additional plane and rail that Fit Fly has over Slim Fly.

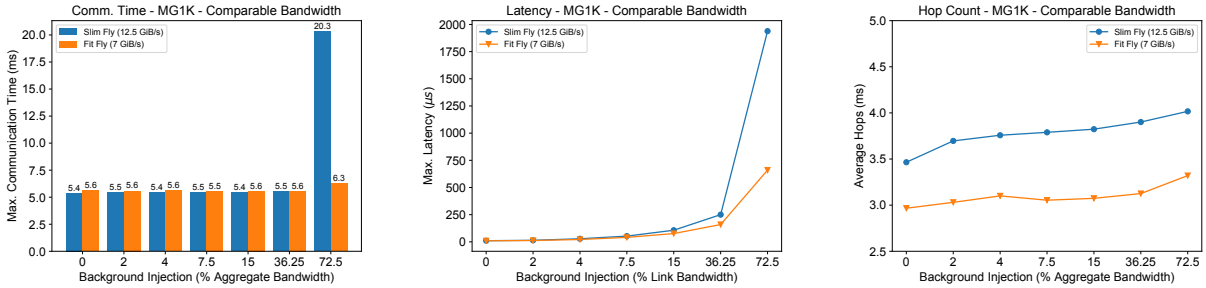
3.5 Discussion

The demand for stronger performance from HPC systems continues to grow. Building exascale systems and overcoming the challenges associated with that endeavor will require a



(a) (b) (c)

Figure 3.6: Synthetic interference experiments showing (a) communication time, (b) packet latency, and (c) hop count on the AMG1728 trace workload with 1,000 synthetic background ranks. Link bandwidth of Slim Fly in this case is 12.5 GiB/s (≈InfiniBand EDR) while Fit Fly is 7 GiB/s (≈InfiniBand FDR). Total aggregate bandwidth is calculated by $B_L \cdot P$, where B_L is the bandwidth of each link in the network.

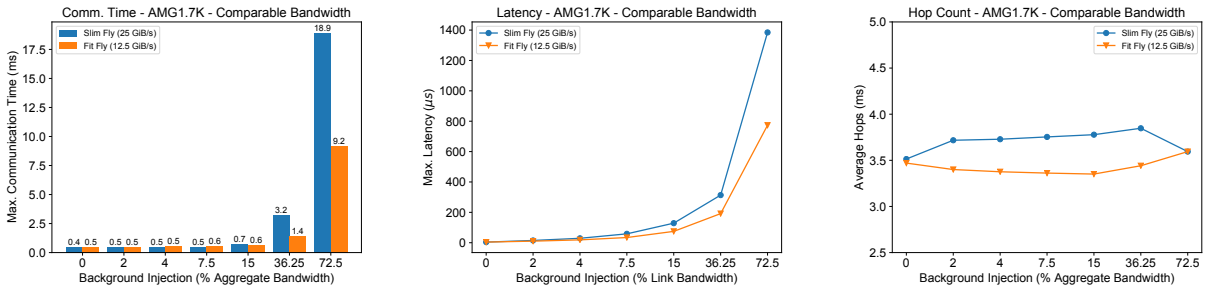


(a) (b) (c)

Figure 3.7: Synthetic interference experiments showing (a) communication time, (b) packet latency, and (c) hop count on the MultiGrid1000 trace workload with 1,000 synthetic background ranks. Link bandwidth of Slim Fly in this case is 12.5 GiB/s (≈InfiniBand EDR) while Fit Fly is 7 GiB/s (≈InfiniBand FDR). Total aggregate bandwidth is calculated by $B_L \cdot P$, where B_L is the bandwidth of each link in the network.

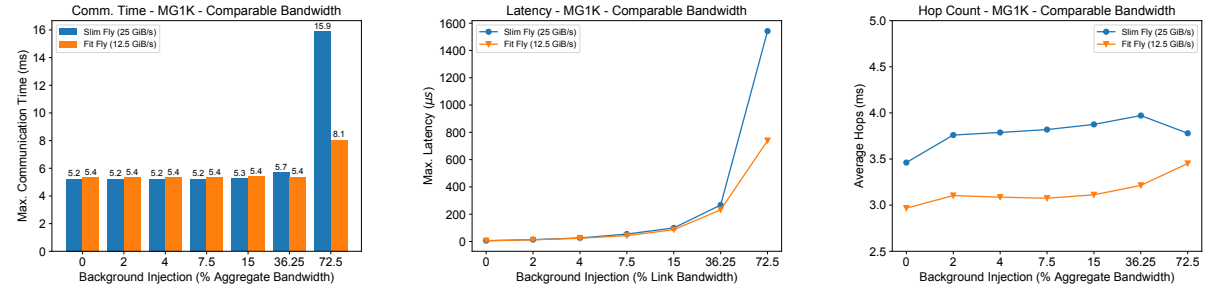
lot of innovative techniques and ideas.

We have performed several experiments in order to answer a "what if?" question. These include comparing the performance of our new implementation with state-of-the-art networks and determining what makes this new model so high-performing.



(a) (b) (c)

Figure 3.8: Synthetic interference experiments showing (a) communication time, (b) packet latency, and (c) hop count on the AMG1728 trace workload with 1,000 synthetic background ranks. Link bandwidth of Slim Fly in this case is 25 GiB/s (≈InfiniBand HDR) while Fit Fly is 12.5 GiB/s (≈InfiniBand EDR). Total aggregate bandwidth is calculated by $B_L \cdot P$, where B_L is the bandwidth of each link in the network.



(a) (b) (c)

Figure 3.9: Synthetic interference experiments showing (a) communication time, (b) packet latency, and (c) hop count on the MultiGrid1000 trace workload with 1,000 synthetic background ranks. Link bandwidth of Slim Fly in this case is 25 GiB/s (≈InfiniBand HDR) while Fit Fly is 12.5 GiB/s (≈InfiniBand EDR). Total aggregate bandwidth is calculated by $B_L \cdot P$, where B_L is the bandwidth of each link in the network.

3.5.1 Comparing with Other Networks

It is valuable to see how a new idea might compare with old ideas. To do so, we compared Fit Fly with its originating model, Slim Fly, as well as two similarly sized Dragonfly-class networks.

The Dragonfly network, for example, is currently slated as the underlying model for the upcoming Cray Slingshot interconnect [56]. Megafly, or Dragonfly+, is also a contender as

the skeleton for future exascale-level HPC systems [37], [40]. The results of these experiments showed that the Slim Fly model itself is competitive with both Dragonfly and Megafly. It matched their performance in both the AMG and MultiGrid trace workloads in low interference studies and outperformed them in higher interference cases. Overall we observed a reduction in maximum communication by to up to 50%.

Fit Fly, with an additional plane of independent routers, showed great resilience to high levels of interference. A 72.5% interference traffic run was performed on the Slim Fly and Fit Fly networks to extend Figures 3.4 and 3.5 but was not included because the Dragonfly-class networks had such difficulty completing the workloads with that level of interference.

We note that occasionally the Fit Fly average hop count for the baseline 0% background traffic was actually higher than subsequent, more intense interference runs. This is counterintuitive and is likely a result of an overly aggressive `CONGESTION` planar selection scheme choosing a worse path to avoid small levels of congestion.

Because the Dragonfly-class networks had such difficulty handling the high levels of interference, we stopped the simulations at 1 second (simulation time). Even if we let the simulations complete, it would be difficult to represent the accurate performance metrics while also maintaining the clarity needed to compare Fit Fly with the other networks. For transparency, we have included the results of those runs in Table 3.3.

Table 3.3: Maximum communication time with synthetic interference injected at 72.5% of link bandwidth (extension of Figures 3.4 and 3.5).

Trace Workload	Slim Fly	Fit Fly	Dragonfly	Megafly
AMG1728	28.4 ms	**1.4 ms**	> 1 s	> 1 s
MultiGrid1000	20.3 ms	**5.4 ms**	> 1 s	> 1 s

3.5.2 Comparing with Slim Fly

As noted in Section 3.4.5, the comparisons made between the Slim Fly and Fit Fly networks following the configurations in Table 3.2 give Fit Fly a distinct advantage in total bandwidth and router count.

The results of the experiments between Slim Fly and Fit Fly with comparable aggregate bandwidths show that bandwidth alone is not why Fit Fly performs so well compared with its single-rail-single-plane counterpart. Looking at the average number of hops traversed by packets in the networks of Figures 3.6, 3.7, 3.8, and 3.9, we observe that Fit Fly consistently

has a lower hop count than its Slim Fly counterpart has.

With the aggregate bandwidth made comparable between the two networks, the only other difference is the total number of routers from the second plane. The higher number of routers means that there is a lower chance of any two packets sharing a buffer on the same router. Because of the increased number of routers, Fit Fly is able to keep the average number of hops that packets traverse lower, typically, than that of Slim Fly and thus is capable of handling greater levels of traffic even with slower, less expensive links.

We note that the performance of the reduced link bandwidth Fit Fly is far better than that of Slim Fly at high levels of interference in these experiments but is beaten slightly by Slim Fly at lower levels of interference. We suspect that this is a result of CODES using link bandwidth for the speed of transmission of data from the workload server and the terminal node instead of the total injection bandwidth.

We also note that at extreme interference workloads, the average hop count sometimes decreases. We believe this is a result of the primary workload finishing and the remaining queued packets from the background traffic working their way through the network without as much contention.

3.5.3 Cost Comparison

Cost is an important consideration when comparing Fit Fly with the other networks. To give an approximation of how much each of the networks tested in this work would cost to build, we have created a virtual shopping cart of network switches and links using Mellanox's online store [57]. The bandwidths configured for the networks in Table 3.2 equate to Mellanox IB EDR-class interconnect hardware. Additionally, we want to consider whether the reduced-bandwidth Fit Fly network could be a decent compromise between expense and performance using cheaper Mellanox IB FDR-class hardware.

In Table 3.4, we give the estimated cost to construct the various networks in this chapter—excluding a Slim Fly HDR system since the Mellanox online store had no official quote for that hardware at the time of writing.

Slim Fly, with its lower number of links and routers, wins the "least-expensive" award, followed by Dragonfly, Megafly, and finally Fit Fly. It is unsurprising that Fit Fly loses a flat cost comparison. However, even the less expensive Fit Fly FDR system outperformed all the other networks—enough so that even cheaper interconnect hardware could still potentially

Table 3.4: Cost breakdown of the various networks tested.

	Slim Fly EDR	Fit Fly EDR	Fit Fly FDR	Dragonfly EDR	Megafly EDR
Links	6,253	12,506	12,506	7,524	8,100
Routers	338	676	676	342	360
Cost per Link	$134	$134	$86	$134	$134
Cost per Router	$25,633	$25,633	$19,803	$25,633	$25,633
Total Link Cost	$837,902	$1,675,804	$1,075,516	$1,008,216	$1,085,400
Total Router Cost	$8,663,954	$17,327,908	$13,868,828	$8,766,486	$9,277,880
Total Network Cost	**$9,501,856**	**$19,003,712**	**$14,462,344**	**$9,774,702**	**$10,313,280**

outperform the state of the art without incurring too much expense.

With the strong performance of Fit Fly taken into consideration, the additional costs of Fit Fly (which can be reduced while maintaining good performance by using less expensive hardware) could be worth considering for many HPC system integrators.

3.5.4 Parallel Simulation Advantages

All experiments in this chapter were performed by using the ROSS and CODES simulation frameworks. A primary feature of this simulation system is its built-in capability for parallelization. Large simulations stand to benefit greatly from the added processing power to handle the increased number of events.

To qualitatively show the type of speedup gained from running simulations in parallel, we performed the simulations in Section 3.4.4 in both sequential and parallel across 24 processing cores on our 32-core Intel Xeon E5-2640 v3 CPU server at 2.60 GHz with 110 GB of total RAM. Full scaling analysis of ROSS, CODES, and the CODES base Slim Fly model, which Fit Fly is iterated from, have been performed in previous works [24], [50], [58]–[60] and was not performed for this work due to time and space constraints.

The results of this comparison are shown in Figure 3.10. Optimistic execution granted great speedup (≈7x) in these simulations, which allows for rapid feedback. The longest-running simulation overall was the 72.5% interference run on the 25 GiB/s configured Slim Fly network in Figure 3.8a, which came in at 99,478 seconds in optimistic mode. For the sake of time, a sequential run was not attempted for comparison.

Parallel simulation even has an advantage over running multiple sequential simulations at once in the form of reduced overall memory consumption. A single parallel simulation largely needs only to allocate extra memory for event buffers for each PE, while multiple sequential simulations re-instantiate the entire simulation for each process.

ROSS+CODES is flexible and allows researchers to choose the execution mode that best suits their needs.

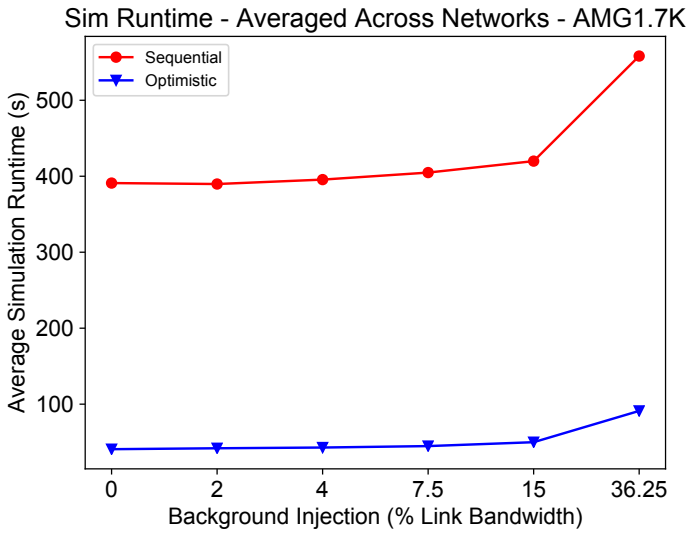


Figure 3.10: Simulation runtime comparison between sequential and optimistic (parallel) execution. Times are averaged across the four networks tested in Section 3.4.4. Sequential simulations are executed on a single PE while the optimistic simulations were spread across 24 PEs.

3.6 Conclusion

To summarize, we present Fit Fly, an enhanced Slim Fly simulation model that enables the use of multiple independent router planes and multiple rails for packet injection. To facilitate proper utilization of these added features, we have included three schemes for dictating which rail and consequently which plane any given packet will be injected on.

We include a method to increase overall path diversity of the Fit Fly network by providing additional rules for how terminal nodes are connected to routers on each plane.

The new network model was validated against the previously validated Slim Fly network model by injecting varying levels of traffic into the Fit Fly network. We observe that the accepted load of the network scales with the total number of rails and planes.

We performed multiple experiments comparing four similarly sized networks with one another with two HPC application traces with varying levels of background interference traffic. In these experiments, the strong behavior of Slim Fly and Fit Fly models was noted

with a distinct lead by the Fit Fly model.

Arguably the comparison of the Fit Fly model may not have been completely fair because the total bandwidth of Fit Fly was double that of Slim Fly. To account for this situation, we performed experiments with Slim Fly and Fit Fly models where the total bandwidth of each network was made comparable. Even with reduced Fit Fly bandwidth or increased Slim Fly bandwidth, the Fit Fly model had greater resilience to high levels of interference traffic. This resilience is observed in application performance and average packet latency and the average number of hops traversed by packets in the network, indicating that the network did not have to divert packets on longer paths to avoid congestion hot spots.

The additional power of Fit Fly does come at extra cost, however. Nevertheless, we have shown that the increased network resilience may be worth the cost of the additional router planes and that increased power can be achieved through utilizing less expensive lower bandwidth links.

We have shown that the CODES+ROSS framework can be a valuable tool to test new ideas such as routing algorithms, load-balancing schemes, and other prototypical features. Insight can be gained by creating new or enhancing existing network models in the CODES model lineup.

In future work, we would like to experiment with other ways to utilize additional rails and planes, such as a quality of service implementation at the planar selection level. Another interesting venture would be to implement these same multi-rail-multi-plane enhancements in other network models.

Testing new concepts and gaining real-world-scale predictions of how they may perform is crucial to continuing innovation in the field of interconnection networks. As congestion avoidance mechanisms grow in complexity, accurately predicting network behavior via mathematical analysis will become more challenging. High-fidelity parallel simulation frameworks such as CODES give researchers the tools necessary to model, test, and simulate concepts at full scale in an efficient manner.

CHAPTER 4
EVALUATION OF LINK FAILURE RESILIENCE IN MULTI-RAIL DRAGONFLY-CLASS NETWORKS

During long-term operation of a high-performance computing (HPC) system with thousands of components, many components will inevitably fail. The current trend in HPC interconnect router linkage is moving away from passive copper and toward active optical-based cables. Optical links offer greater bandwidth maximums in a smaller wire gauge, less signal loss, and lower latency over long distances and have no risk of electromagnetic interference from other nearby cables. The benefits of active optical links, however, come with a cost: an increased risk of component failure compared with that of passive copper cables.

One way to increase the resilience of a network is to add redundant links; if one of a multiplicity of links between any two routers fails, a single hop path will still exist between them. But adding redundant links comes at the cost of using more router ports for router-router linkage, reducing the maximum size of the network with a fixed router radix. Alternatively, a secondary plane of routers can be added to the interconnect, keeping the number of compute node endpoints the same but where each node has multiple rails of packet injection, at least one per router plane. This multi-rail-multi-planar type of network interconnect allows the overall size of the network to be unchanged but results in a large performance benefit, even with lower-specification hardware, while also increasing the resilience of the network to link failure.

We extend the CODES framework to enable multi-rail-multi-planar 1D-Dragonfly and Megafly networks and to allow for arbitrary link failure patterns with added dynamic failure-aware routing so that topology resilience can be measured. We use this extension to evaluate two similarly sized 1D-Dragonfly and Megafly networks with and without secondary router planes, and we compare their application communication performance with increasing levels of link failure.

This chapter previously appeared as: N. McGlohon, R. B. Ross, and C. D. Carothers, "Evaluation of link failure resilience in multirail dragonfly-class networks through simulation," in *Proc. ACM SIGSIM. Conf. Princ. Adv. Discrete Simul.*, 2020, pp. 105–116.

4.1 Introduction

Simulation of current and prototypical high-performance computing (HPC) interconnect networks is a useful tool for generating realistic performance expectations of proposed system acquisitions. Different interconnect designs, or network topologies, have varying strengths and weaknesses with different price-to-performance ratios. As mentioned in previous chapters, the cost of building a full-scale HPC interconnect is prohibitive for the purposes of evaluation. Using simulation, one can develop models for any arbitrary network definition and compare them with other, previously defined models.

This flexibility allows for fast, realistic predictions of how different topologies or network technologies may perform. With simulation, one can answer questions about how different network types behave at different scales, how different routing algorithms perform, or how job placement affects performance. It also allows for testing how the system might behave in states of irregularity, such as in the case of inter-router link failure.

Link failure is a common occurrence in HPC systems—so common in a large-scale system that between monthly maintenance routines, multiple faulty links may occur between routers in the network. Thus, system builders need to know that the interconnect they are purchasing is capable of still operating despite common link failure. This knowledge allows them to stick to routine maintenance of a month or more so as to minimize downtime caused by the link failure, as opposed to immediately bringing parts of the system down for emergency system repair.

Routing algorithms for HPC interconnects conventionally take certain characteristics of networks as a given, and virtual channel (VC) assignment can be made appropriately to avoid deadlock. Introducing even a single link failure into the network can potentially reduce to zero the number of legal, deadlock-free paths for certain routing algorithms. Also, routing algorithms that could correctly explore all possible legal routes in typical topologies may fail to do so in the presence of link failure. Irregular systems—those with abnormal, arbitrary, nor uniformly defined structure—require additional attention when routing in order to avoid cyclic channel dependency problems such as head-of-line blocking or deadlock [29], [61].

Because of the need to avoid channel dependency problems, one cannot assume that router validity is assured just because the network is connected. A legal path between two endpoints in an interconnect is any path that does not cause cyclic channel dependencies. Existing routing algorithms, such as those in 1D Dragonfly and Megafly, must be altered to

ensure that only legal paths are taken in the presence of link failures. If a valid legal path does not exist, then communication (and the application making it) should fail. Determining whether a legal path exists is a computationally complex problem because the enumeration of all possible paths grows significantly as the number of endpoints (and, by association, routers) increases.

We extend the CODES [62] interconnect network simulator to allow for links in simulated networks to be marked as failed and for packets to be routed dynamically around failed links along legal paths, if and only if they exist. If all legal paths for a packet between two endpoints have at least one failed link, then communication fails, and the simulation ends.

The network types evaluated in this chapter are the 1D-Dragonfly and Megafly networks. 1D Dragonfly, like other Dragonfly-class networks, has groups of routers in which each group is connected to every other group with at least one global link. Within these router groups, the 1D Dragonfly uses an all-to-all configuration for local router links.

Megafly, also known as Dragonfly+, has the same global link structure as 1D Dragonfly has; but instead of all-to-all links within local router groups, it uses a 2-level fat tree configuration to form a fully-connected bipartite graph. Router responsibilities are also divided differently in Megafly. Routers with global links, known as spine routers, have no compute node terminal links. The other routers, those with compute node terminal links but no global links, are known as leaf routers. This type of link configuration is designed to increase the scalability of a network with a lower cost [15].

In this chapter, we study the effects of link failure in Dragonfly-class topologies and posit a strategy for increased resilience—multi-planar networks.

The main contributions of this chapter are as follows:

1. Introduction of a *network manager* to the CODES framework to orchestrate link failure, and identification of legal paths given virtual channel constraints

2. Extension of the CODES Dragonfly and Megafly routing algorithms— minimal (MIN), Valiant non-minimal (VAL), and progressive adaptive (PAR)—to route dynamically in the presence of network link faults

3. Extension of the CODES Dragonfly and Megafly network models to support multi-rail-multi-plane configurations with multiple schemes for injection rail selection

4. Evaluation of multi-rail-multi-planar inter-job interference performance and the effect of multi-planar configurations on the resilience of two example Dragonfly and Megafly networks with varying levels of link failure

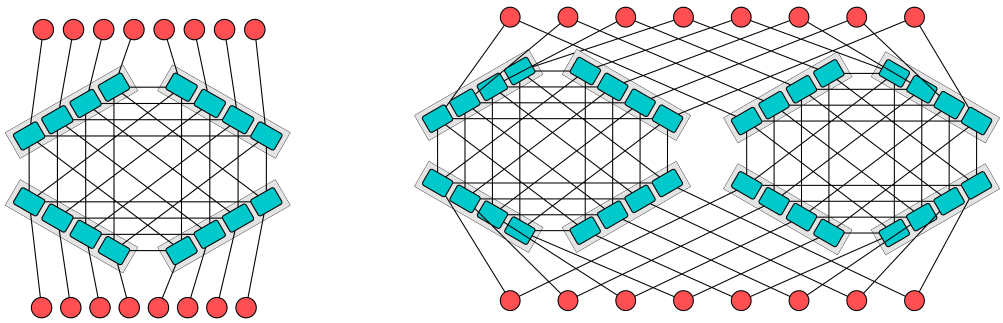
4.2 Background

We study the effects of link failure in HPC network interconnects made up of routers or switches connected in a specific pattern known as a network topology. Also connected to these routers are compute node terminals that inject packets into the network to be routed along paths to other compute nodes. In a real-world interconnect, these packets are the carriers of information between processes in an HPC application. We can use captured real-world communication patterns of HPC applications for simulated packet generation or create our own synthetic jobs to act as benchmark workloads in simulated networks. By simulating multiple simultaneous workloads, we can analyze how varying intensities of communication from different workloads interfere with each other. Higher levels of interference reflect poorer network performance. More information regarding simulation background can be found in Chapter 1.4.

4.3 Network Model

The network topologies examined in this chapter are the 1D-Dragonfly and Megafly networks. Dragonfly-class networks have one particular characteristic in common: routers in the network are grouped together into local groups with some degree of local link connectivity between them. Global link connections, those that span between two local groups, are configured such that at least one global link exists between any two pairs of local groups. In other words, local groups themselves are connected all-to-all.

How these topologies differ lies in how local groups are configured. In 1D-Dragonfly, routers within a local group are connected in an all-to-all fashion. This makes the assured minimum number of hops between any two routers in a local group equal to one. When router groups become large, however, this results in a greater cost because of the increased number of local links. Because the groups are also mandated to have at least one level of all-to-all connectivity between them, increasing the number of groups with the aim of reducing the number of routers per group will result in an increase in the cost associated with global links.



(a) (b)

Figure 4.1: Example layout of single-rail-single-plane (a) and dual-rail-dual-plane (b) 1D Dragonfly networks with four routers (blue squares) per group and four groups per plane. There are 16 compute node hosts (red circles) attached to each plane in a given network. In the single-rail-single-plane case, each node has one rail for injection to a router (blue squares), but in the dual-rail-dual-plane case, each node has two rails for injection, one per plane. The high-level connection scheme of Megafly follows similarly with the exception that routers within local groups are connected in the shape of a fully-connected bipartite graph instead of all-to-all like 1D Dragonfly.

In contrast, the Megafly network topology has routers within local groups connected in a two-level fat-tree configuration. As a result, these local router links form a fully-connected bipartite graph of two types of routers: *leaf* routers, which have local router and terminal/compute node links, and *spine* routers, which have local router and global router links. No terminals are attached to spine routers, and no leaf routers own any global links.

4.3.1 Multi-Rail-Multi-Planar Networks

This section focuses on networks with multiple, independent planes of routers interconnecting a fixed set of compute nodes. Because compute node hosts are shared between router planes, each node has multiple rails for packet injection, at least one per plane. For simplicity, the term multi-rail is often used because multi-rail-multi-planar networks are a special case of the broader term. We will use multi-rail and multi-planar terms interchangeably, and the number of injection rails per compute node is equal to the number of router planes in a given network in our work.

4.3.2 Benefits of Multi-Planar Networks

Adding additional planes of routers has shown great ability to reduce the effect of inter-job interference even with reduced bandwidth [63]. We posit in this work that the benefits of multi-planar networks lie in the ability not only to reduce the effect of interference but also to reduce the effect of congestion caused by link failures. Additional planes of independent routers multiply the available paths between any two endpoints in the network by a factor proportional to the number of additional planes. Thus, the odds of completely severing all legal paths between any two endpoints are reduced.

4.4 Link Failures

We have extended the CODES interconnection simulation framework to include an organizational class called a *network manager*. This class stores all link connection information for a network, including the failure state of the links. This data is loaded in at the start of the simulation and is identical across all PEs in the simulation—the state of links with regard to failure is static throughout the simulation. Because the time periods simulated by CODES are generally less than a few seconds, we felt that enabling in situ link failure was not crucial. Additionally, as observed in this work, the effect of a single link failure (or any number that would reasonably be expected to occur in the course of seconds) is generally insignificant to application performance.

In a realistic mid-simulation link-failure scenario, a management communication layer may be implemented to facilitate the transmission of global knowledge. Adapting the CODES network manager to allow for this would simply require additional ROSS events to facilitate the transmission of knowledge to all LPs. Allowing for in situ link failures, however, brings in new layers of uncertainty. For example, 'what happens to a packet that was already injected prior to link failure and its only legal path became severed? How this error is handled depends on the implemented communication standard and is outside the scope of this study. Here we assume that information regarding link failure is a priori knowledge known by all terminals and routers in the network.

The network manager provides an interface for routers to query for information about the network. For example, a router has a connection manager subclass of the network manager that has information about connections specific to it and its group and can answer questions such as the following: What routers am I connected to locally, and what routers

am I connected to that have a connection to a specific group? An additional parameter can be added to these queries to take into consideration whether a link has failed.

The addition of link failures to a network model poses challenges. In CODES, most routing algorithms are implemented to take advantage of given knowledge about the network. For 1D Dragonfly, one of these bits of design knowledge is the expectation of having at least one global link between any two groups.

Imagine a 1D-Dragonfly network configured so that each group is connected to every other group with exactly one global link. Now picture a packet being generated in a group A and destined for a router in group B. In a minimal path, it will need to traverse exactly one global link; if that global link has failed, then the only minimal path between any routers in group A and group B no longer exists. Therefore, it would need to be routed nonminimally through a router in an intermediate group. However, we also have to be aware of any failures in a potential intermediate group. The only global link between the intermediate group and group B may also have failed, and hence that group would be ineligible for a nonminimal route between groups A and B.

Unfortunately, there are limitations to how packets can be routed in these interconnects so as to avoid the performance-limiting issue of head-of-line blocking or the performance-halting issue of deadlock [61]. Virtual channels, a method of virtually partitioning router ports into subunits, are commonly used in Dragonfly networks to avoid these issues by requiring that specific rules be followed about what VCs to assign a packet to when routing. In order to avoid deadlock, it is crucial to avoid what is referred to as a cyclic channel dependency [29], [64].

In order to avoid these channel dependency issues, rules are imposed on the routing algorithms that set the maximum number of hops allowed to be traversed within each group and how many global hops may be utilized in routing a packet from group A to group B. These constraints mean that though a router is technically connected to the network, it still may not be legally accessible by any or all other routers in the network.

If a packet is injected into a router plane and the routing algorithm fails to find a legal path, or if a legal path does not exist, then the packet will reach a dead end, and the workload will fail. Optimally, then, we would like to inject a packet only onto a router plane that has a valid path to its destination. To do so, we need to be able to answer two questions.

1. Does a legal path exist between source router x and destination router y with at most l_s

local hops in the same group of source router, at most l_i local hops in any intermediate group, at most l_d local hops in the destination group, and exactly g global hops?

2. Given that a legal path matching these criteria exists, what are the next stops from router x that would match any path matching the criteria?

Using constraints of the Dragonfly and Megafly networks, notably that l_s, L_i, L_d equal 1 and 2 for Dragonfly and Megafly, respectively, we can simply use l local hops per group in the problem formulation instead of three separate variables. If we had a black-box solution to the first question, we could know whether there exists a minimal path within a local group ($g = 0$), a minimal or near-minimal path with no intermediate groups ($g = 1$), or a legal nonminimal route ($g = 2$) with no need to adjust any VC assignment strategies in the network model.

4.4.1 Algorithmic Solution

We have developed an algorithm for solving these problems in Dragonfly-class networks leveraging the known constraints to reduce the algorithmic and computational complexity. This problem can be solved via a dynamic programming-like approach with memoization/caching to quickly return an answer to a query.

Algorithm 1 solves the problem of finding the existence (and the next hops) of a legal path defined by l allowed local hops in the current group and L maximum local hops in any group visited and with exactly g global hops. It is a modified depth-first-search algorithm to traverse a graph to find if the destination router y is reachable by x with the given restraints.

Requiring exactly g global hops allows us to narrow down what types of paths will be included in our query. If we allowed it to return paths with at most g hops instead, then we would not be able to prevent nonminimal intermediate group (2 global hops) paths from being included in the query response. For the progressive adaptive algorithm, we need to be able to compare minimal and nonminimal paths in order to decide how to route to the final destination.

The algorithm uses recursion, knowing that if there is a valid local hop from the current router with the given restraints, then there must first be a valid hop from a locally connected router with one less local hop in the current group. Similarly, if there is a valid global hop from the current router with the given restraints, then there must first be a valid hop from a

Algorithm 1 Finding Existence and Next Stops in Legal Path

Memoization Map of Input to Response: $M(input)$
Maximum Local Hops Allowed per Group: L

Input i:
    Source Router GID: x
    Destination Router GID: y
    Local Link Adjacency List for Source: A_l
    Global Link Adjacency List for Source: A_g
    Max Local Hops in Current Group : l
    Exact Global hops in path: g

GetValid(i):
$V_{next} \leftarrow$ Empty set of valid next hops
if $x == y$ and $g == 0$ **then**
    $V_{next} \leftarrow y$
    return V_{next}
if $i \in M$ **then**
    return $M(i)$
if $l > 0$ **then**
    for $link \in A_l$ **do**
        if $link$ is not failed **then**
            $V \leftarrow$ GetValid($link.dest$, y, $l - 1$, g)
            if size(V) > 0 **then**
                $V_{next} \leftarrow link$
if $g > 0$ **then**
    for $link \in A_g$ **do**
        if $link$ is not failed **then**
            $V \leftarrow$ GetValid($link.dest$, y, L, $g - 1$)
            if size(V) > 0 **then**
                $V_{next} \leftarrow link$
$M(i) \leftarrow V_{next}$
return V_{next}

globally connected router with exactly one less global hop in the path—but the number of local hops allowed in the next current group is reset in the subproblem because a global hop was taken.

Because this algorithm only explores potential next stops if the link to the next stop is not failed, this algorithm will not return any paths that would require a failed link to be traversed. If there are no valid next stops with the given constraints returned from the algorithm, then no legal path exists with those constraints.

4.4.2 Algorithmic Caveats

Traditional depth-first search requires a set of visited graph nodes to prevent cycles. Instead, we rely on the constraints to ensure legal paths that do not needlessly route.

With 1D Dragonfly, local group hops are limited to one, which makes cycles or round-about paths within a local group impossible. In Megafly, local group hops are limited to 2; but to prevent issues of VC dependency deadlock, all nonminimal paths through an intermediate group must travel through a leaf router, requiring explicitly two local group hops. This means that a 2-hop detour revisiting the same spine router A, as in the example, is mandated by the network model.

One might wonder, then, why not just utilize known, efficient shortest-path algorithms? Algorithms such as Floyd-Warshall All Pairs Shortest Path [65] can find the shortest path between any two nodes in a graph quickly and optimally. We have yet to find a modification of this algorithm that can work with the necessary constraints in order to ensure that the shortest paths found by the algorithm are also the shortest *legal* paths.

One can enumerate all shortest paths and then filter out only legal paths with this method, but doing so requires a large space and time complexity for larger networks. And if we wanted to find paths of at least L hops, as is the case for nonminimal routes used by adaptive routing, we would need additional complexity.

Algorithm 1, however, is not designed to find only the shortest paths. This algorithm could be modified to enable that by encoding the current hop count as input and returning the hop count when the destination is found. Such a modification would come at great cost to computational efficiency however, since it would render memoization impossible: the first found legal path (and thus the memoized response) is not guaranteed to be the shortest. Another way to ensure only shortest paths would be to employ a breadth-first search approach but, again, at the sacrifice of memoization and its computational benefits.

4.4.3 Routing and Planar Selection

To route correctly through a network with link failures, we leverage Algorithm 1 to find legal paths with specific constraints in the path. Most importantly, we specify the exact number of global hops g in a desired path. This allows us to know whether there exist local-group-only paths, single-global-hop paths, or two-global-hop paths.

Enabling CODES to find legal paths given specific constraints allows it to dynamically

determine whether a legal path exists and to route accordingly. If enough links fail, then there may not be any legal paths between a source and destination router; and, if any compute nodes attached to the routers wish to communicate with each other, the communication will fail.

This chapter allows for the 1D-Dragonfly and Megafly models to feature multiple router planes servicing the same set of compute node terminals. Nodes have a choice of what rail to inject a given packet onto. To ensure greater resilience to link failure, we want to inject a packet onto a plane only if we know that there is a legal route between the source and destination compute nodes on that plane. Algorithm 1 gives us the ability to know this prior to injection.

Given a set of V valid planes for injecting a given packet, we have a variety of options for deciding the preferred plane of injection. Following similar work of enabling multi-rail-multi-planar Slim Fly networks in CODES [63] and Chapter 3, we have implemented the following schemes for injection plane selection, with a few modifications to account for link failures.

***PATH* Planar Selection:** The Floyd-Warhsall shortest path evaluation to estimate the distance between two routers can be utilized to minimize the distance traveled by an injected packet. It may not mean that a legal path with that distance exists, but it can generally determine whether significant link failures would require a great deal of routing around.

***CONGESTION* Planar Selection:** This scheme estimates the congestion being experienced by the number of packets set for injection along each rail at the originating terminal. This estimation is made with information from a credit-based flow control system that allows terminals to inject on rails with less inferred back pressure. This is the scheme utilized in the evaluation of multi-planar networks in this chapter.

***RANDOM* Planar Selection:** This scheme involves trivially selecting a valid plane at random. This naively balances load across available planes but does not take into account any congestion that may be present in the network at different levels on different planes.

***DEDICATED* Planar Selection:** This scheme allows for the workload to select the injection rail during the creation of the message. The network makes no alteration. If a valid path on the given plane does not exist, then the workload will fail.

4.5 Validation

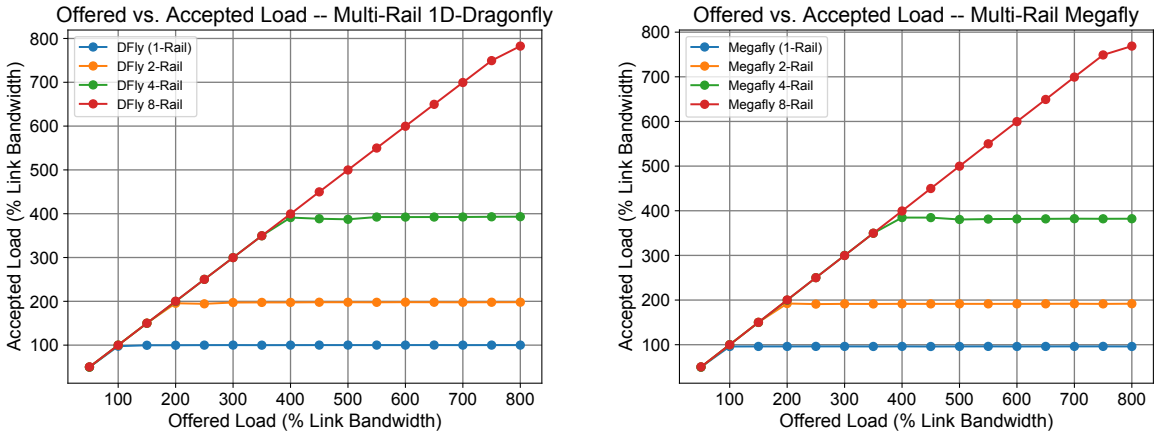
Without access to a physical multi-planar network, truly validating findings from simulation is difficult. We instead rely on other factors to ensure that adding additional rails and planes behaves in the way that we expect.

All routing adjustments for enabling link failure were compared with previous, accepted routing implementations and return results in line with expectations.

We ran a bandwidth-testing sanity check of our implementation of multi-rail-multi-planar 1D Dragonfly and Megafly to ensure that the accepted load of the network scales with the number of planes in the network. If a single plane accepts some load λ, we expect two planes to accept $2 \times \lambda$. Specifically, we constructed four networks of each type, 1D Dragonfly and Megafly, with varying numbers of router planes: 1, 2, 4, and 8. We scaled synthetic uniform random traffic injected into the network at a percentage of the link bandwidth used in the network, ranging from 50% to 800%. Figures 4.2a and 4.2b display the results of the tests. We measured accepted load by counting the number of bytes received by terminals during a given time window and dividing by the length of the time window. Results are in line with expectations: greater numbers of router planes allow for greater amounts of accepted load.

Table 4.1: Configurations of 1D Dragonfly and Megafly networks utilized in this work for performance and resilience comparisons. Note that in some experiments, dual-rail configurations were written to use reduced 7.0GiB/s bandwidth as per Infiniband FDR specifications.

	1D Dragonfly	1D Dragonfly - Dual Plane	Megafly	Megafly - Dual Plane
Router Radix	36	36	36	36
Planes	1	2	1	2
Rails	1	2	1	2
Groups	19	38	10	20
Node Count	3078	3078	3240	3240
Router Count	342	684	360	720
Nodes per Router	9	9	18 (Leaves only)	18 (Leaves only)
Global Connections	3078	6156	3240	6480
Global Conn. / Group	162	162	324	324
Global Connections Between Groups	9	9	36	36
Link Bandwidth	12.5 GiB/s	12.5 GiB/s (EDR) 7.0 GiB/s (FDR)	12.5 GiB/s	12.5 GiB/s (EDR) 7.0 GiB/s (FDR)
Routing Algorithm	Adaptive (PAR) [66]	Adaptive (PAR)	Adaptive (PAR)	Adaptive (PAR)
Planar Selection	n/a	`CONGESTION`	n/a	`CONGESTION`



(a) **(b)**

Figure 4.2: Single-, dual-, quad-, and octo-rail versions of 1D Dragonfly (a) and Megafly (b) were offered varying loads of synthetic uniform random data, sweeping the mean amount of time between when packets of fixed size were injected into the networks. The number of router planes in each network strictly equals the number of rails. Offered and accepted loads are shown here as a percentage of the link bandwidth (12.5 GiB/s).

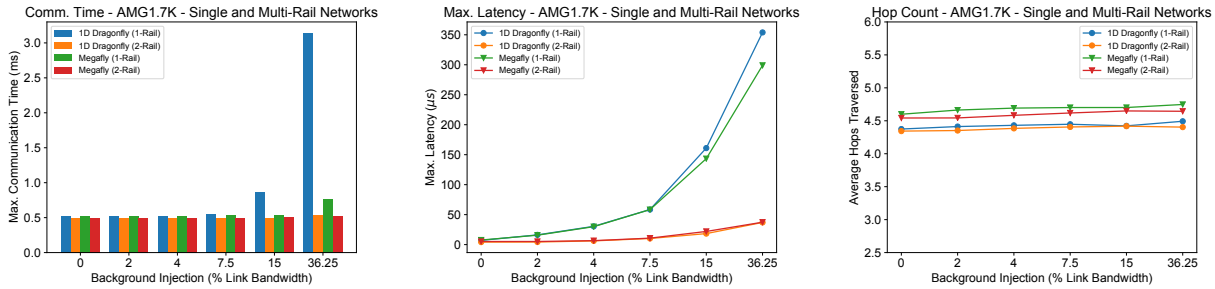
4.6 Experiments

CODES works with DUMPI application traces through the SST DUMPI Trace module [30]. These traces are captured from communication patterns of real HPC applications and grant far greater understanding of how real-world applications might perform on given networks than what is typically achievable through basic synthetic workloads.

We used publicly available DUMPI traces provided by the Design Forward Program [31] of NERSC. All experiments in this chapter were performed on our 40-core Intel Xeon CPU E5-2640 laboratory server. Most simulations completed on the order of minutes run in parallel using ten processor cores each.

4.6.1 Workloads

Algebraic MultiGrid Solver (AMG) Workload: AMG is an algebraic multigrid solver mini-app for unstructured mesh physics packages. Our experiments used the version with 1,728 MPI ranks. We mapped each rank to its own terminal node in the given network. The application from which this trace was created had a runtime of 2.2 seconds, including computation time, with 1,728 MPI ranks across 72 hosts. About half of the time (52%) was



(a) **(b)** **(c)**

Figure 4.3: Synthetic interference experiments showing (a) communication time, (b) packet latency, and (c) hop count on the AMG1728 trace workload with 1,000 synthetic background ranks. Each synthetic rank injects packets into the network at a percentage of the total link bandwidth of 12.5 GiB/s (≈InfiniBand EDR). The same workloads and job allocation maps were used across each network.

time spent on communication [31].

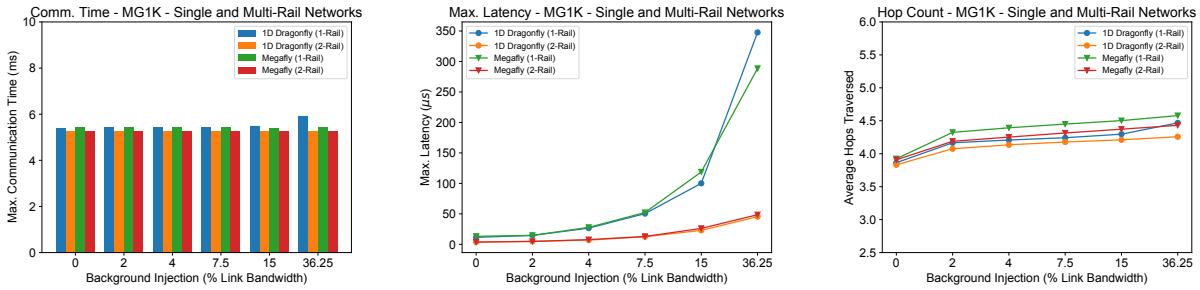
MultiGrid (MG) Workload: MultiGrid is a geometric multigrid cycle mini-app from the adaptive mesh refinement application framework BoxLib [55]. Our experiments used the version with 1,000 MPI ranks, each of which, just as we did for our AMG workload, was mapped to its own terminal node in the simulated network. The application from which this trace was created had a runtime of 1.4 seconds, including computation time, with 1,000 MPI ranks across 42 hosts. The application spent very little (3.7%) of the time on communication [31].

Synthetic Workload: To simulate other random jobs in the network, we injected random traffic using a uniform random synthetic traffic generator. We fixed the mean interval time at the CODES default of 100 μs but increased the overall payload of injected messages to vary the intensity of interference.

4.6.2 Evaluation Metrics

Higher levels of link failure with high levels of injected traffic can result in congestion. We seek to determine how congestion-caused inter-job interference is affected by increasing levels of link failure.

To evaluate the resilience of these networks, we used various application performance metrics, including the average number of hops traversed by packets, the mean latency of all packets in the simulation, and the maximum communication time (the maximum amount of



(a) (b) (c)

Figure 4.4: Synthetic interference experiments showing (a) communication time, (b) packet latency, and (c) hop count on the MultiGrid1000 trace workload with 1,000 synthetic background ranks. Each synthetic rank injects packets into the network at a percentage of the total link bandwidth of 12.5 GiB/s (≈InfiniBand EDR). The same workloads and allocation maps were used across each network.

time spent by any one primary workload process participating in the workload). These metrics can be measured on networks with a range of link failures and can be directly compared with networks without link failures.

4.6.3 Studies Performed

The work in this chapter combines a standard interference study of single- and dual-plane 1D-Dragonfly and Megafly networks with a study of interference on the same networks with varying degrees of link failure.

In the link failure study, we also looked at how the same level of link failure affects a reduced-bandwidth version of the dual-planar network configurations. The aim is to show how the additional costs of building multiple router planes can be mitigated by using cheaper hardware while maintaining performance benefits similar to those of the standard multi-plane configurations as noted in [63].

We failed links uniformly at random across the entire tested network until the desired percentage of failed links was met. Information about failed links was generated by using a script and was loaded by the CODES simulation.

4.7 Discussion

In this section, we first present an inter-job interference study as a baseline comparison of single- and dual-plane 1D-Dragonfly and Megafly networks with no link failures. We then,

in Section 4.7.2, present a network resilience study by taking the highest level of interference observed from Section 4.7.1 and apply it to single- and dual-plane networks with increasing levels of link failure.

Table 4.1 shows the network configurations used. Our tested single-plane networks feature links with bandwidths comparable to InfiniBand EDR (12.5 GiB/s). Our tested dual-plane networks consist of two versions each, one with the same InfiniBand EDR specification links and the other using only InfiniBand FDR equivalent links (7 GiB/s).

4.7.1 Single-Rail to Multi-Rail Comparison

We begin by comparing single- and dual-plane 1D-Dragonfly and Megafly networks with zero-failed links. Our aim is to show a baseline standard of benefit that can be expected from a multi-planar network variation.

Because of the different balancing recommendations for generating these networks, creating a truly fair comparison of the different topologies is difficult. To create a Megafly network with approximately 3,000 compute node terminals with a 36 router radix (to utilize the same hardware specifications as the tested 1D Dragonfly), we must have 36 global connections between any two groups in the network. 1D Dragonfly, on the other hand, has only 9, and thus path diversity for inter-group messages is reduced. Nevertheless, we attempted to generate two networks of similar capabilities.

Because job locality plays an important role in mitigating the effects of high network usage, we tried to choose a job-to-compute-node mapping that is "equally poor" for all networks: namely, random mapping. The same job-to-compute-node mappings were used between all evaluated networks. Unfortunately, because of the greater amount of intergroup linkage in the Megafly network, we still expect some performance bias in favor of Megafly with this mapping.

Figures 4.3 and 4.4 show the results of experiments performed with the AMG1728 and MG1000 workloads with synthetic background interference traffic on the single- and dual plane versions of 1D Dragonfly and Megafly without link failures. Background interference is scaled in these experiments from 0% (primary workload only) to 36.25%. The experiments utilize the same failure-aware routing schemes developed by using Algorithm 1.

Figures 4.3a and 4.4a show that dual-rail networks have greater capacity than do single-rail networks to operate as if uncongested. The network has more places for packets

to exist, and thus, for a given set of packets, the odds of two packets finding themselves in the same router buffer is reduced. Thus, there is less contention for bandwidth, inter-job interference is minimized, and the maximum amount of communication time spent by any one node is decreased at higher levels of background interference.

Following a similar trend, Figures 4.3b and 4.4b show that the maximum packet latency is also significantly reduced in the multi-planar variants. The average hop count traversed by packets in the network, shown in Figures 4.3c and 4.4c, is not greatly affected but does tend to be slightly reduced in multi-planar versions.

4.7.2 Link Failure Study

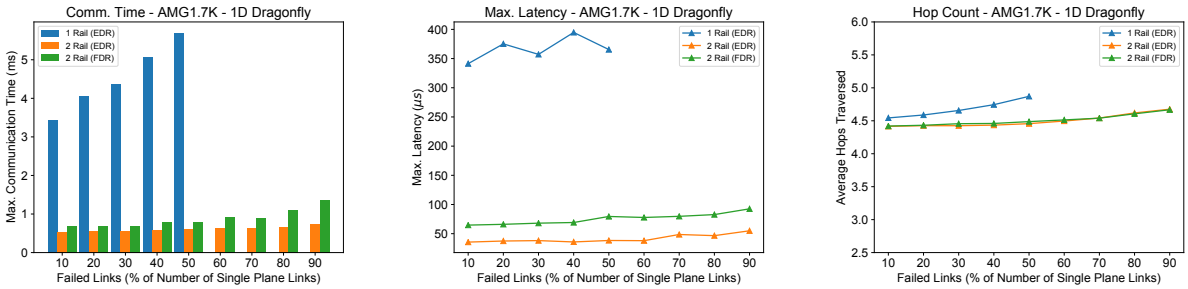
Experiments for this study use the same DUMPI trace workloads, AMG1728 and MG1000, as in Section 4.7.1 but with a fixed level of background interference: 36.25%. Instead of scaling the interference, we scaled the number of failed links as a percentage of the number of links in a single plane. To evaluate how resilient different versions of these networks are, we applied consistent link failure counts to each network type.

We included the FDR variants for consideration because providing an additional plane of routers on top of the first gives a distinct advantage to the multi-planar network in nearly any type of comparison except for cost. The increased cost of multi-planar networks can be mitigated by using lower-specification hardware.

Figures 4.5, 4.6, 4.7, and 4.8 show the results of these experiments. In Figures 4.5a, 4.6a, 4.7a, and 4.8a, we see that as we increase the number of failed links in the network, performance suffers. Failed links reduce the number of available paths for packets to take, and thus packets are far more likely to compete for bandwidth along inter-router links.

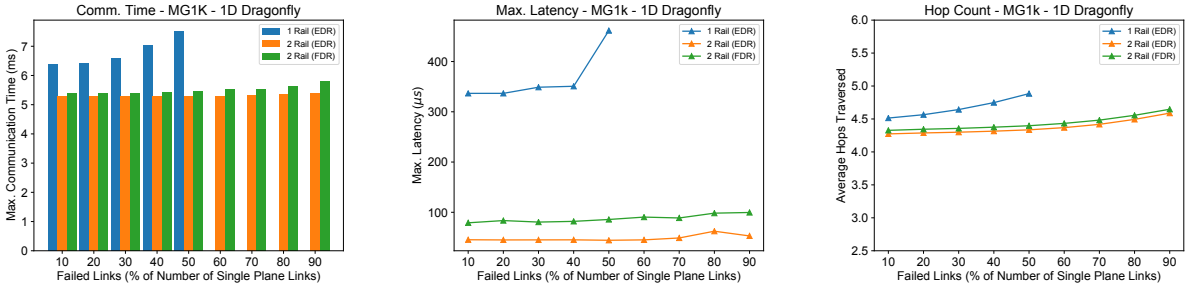
However, as we reach 50% failed links, the single-plane network configurations start to fail since there were no legal paths between select endpoints in the workload. The dual-plane configurations were able to withstand significantly more numbers of failed links before starting to show signs of congestion or interference, let alone failing communication altogether.

AMG is an application that is much more communication-intensive than MG is, with a lot more collective operations that prevent the application from moving forward and injecting new packets into the network until the previous packets complete. Because AMG is collective heavy, its maximum communication time (the maximum amount of time any one process spends doing communication) is more sensitive to interference. Because new packets are not



(a) (b) (c)

Figure 4.5: Link failure resilience study showing (a) communication time, (b) packet latency, and (c) hop count on the 1D-Dragonfly topology with the AMG1728 trace workload with 1,000 synthetic background ranks. Each synthetic rank injects packets into the network at 36.25% of 12.5 GiB/s. Note that the total number of links failed is based on a percentage of the number of links found in a single plane.

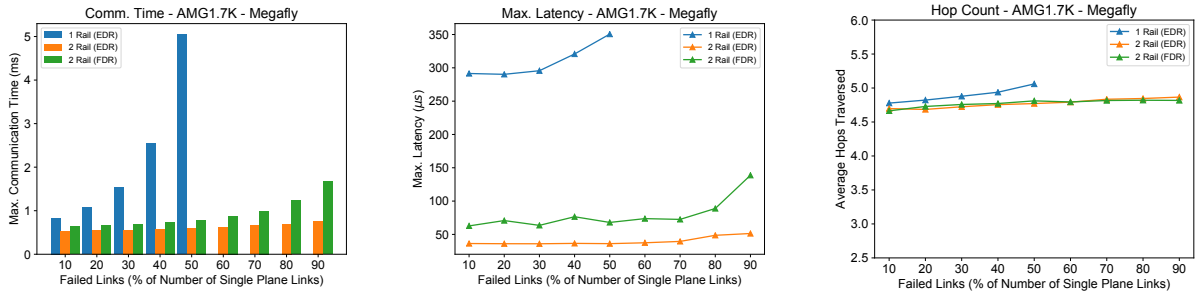


(a) (b) (c)

Figure 4.6: Link failure resilience study showing (a) communication time, (b) packet latency, and (c) hop count on the 1D-Dragonfly topology with the MG1000 trace workload with 1,000 synthetic background ranks. Each synthetic rank injects packets into the network at 36.25% of 12.5 GiB/s. Note that total number of links failed is based on a percentage of the number of links found in a single plane.

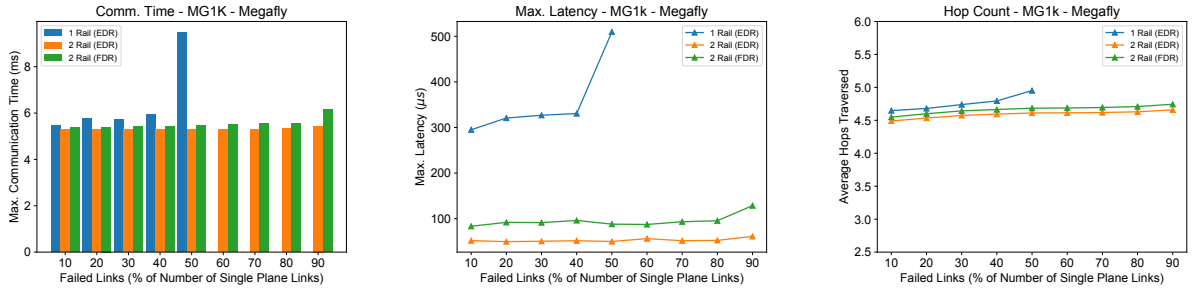
introduced until the previous collectives have completed, the maximum latency (maximum time that a single packet spends in the network) is not as affected. MG, on the other hand, mostly involves MPI_SEND and MPI_IRECV calls, with some MPI_WAITSOME's as well, and so it has fewer operations to prevent it from requesting to inject packets into an already congested network.

Effects of the failed links in dual-plane variations were felt mostly by the lower-specification simulated hardware, while the standard dual-plane versions seem mostly unperturbed by the changes to the network.



(a) (b) (c)

Figure 4.7: Link failure resilience study showing (a) communication time, (b) packet latency, and (c) hop count on the Megafly topology with the AMG1728 trace workload with 1,000 synthetic background ranks. Each synthetic rank injects packets into the network at 36.25% of 12.5 GiB/s. The total number of links failed is based on a percentage of the number of links found in a single plane.



(a) (b) (c)

Figure 4.8: Link failure resilience study showing (a) communication time, (b) packet latency, and (c) hop count on the Megafly topology with the MG1000 trace workload with 1,000 synthetic background ranks. Each synthetic rank injects packets into the network at 36.25% of 12.5 GiB/s. The total number of links failed is based on a percentage of the number of links found in a single plane.

The maximum packet latencies in the experiments are also interesting. Figures 4.5b, 4.6b, 4.7b, and 4.8b show that the dual-plane versions have a distinct advantage with regard to latency. The effects of reduced bandwidth are noticeable but still significantly improved over the standard specification single-plane version.

The experiments also showed the effects of congestion on hop counts. In CODES, the hop count is the number of routers traversed in a path by a packet. We can see in Figures 4.5c, 4.6c, 4.7c, and 4.8c that the networks start to experience some level of congestion as we increase the number of failed links. This is evident by the higher number of hops

traversed by packets. Because of the progressive adaptive routing scheme employed by 1D Dragonfly and Megafly, packets can be routed along longer, nonminimal routes when congestion is detected locally.

We chose to fail a fixed number of links for comparing across single- and dual-plane networks in order to keep as many variables as possible constant in comparisons. Nevertheless, even if one compares, for example, the 80% dual plane data with the 40% single-plane data, the dual-plane network still wins out.

4.8 Future Work

We would like to perform a more thorough evaluation of multi-planar variants of other topologies, including fat tree and Slim Fly.

We would also like to expand on the current link failure study. Fixing the number of link failures and observing how different configurations of networks behave would be helpful for figuring out how best to provision a network.

Another line of work would be to broaden the runs to scale the percentage of failed links with the number of planes so that when comparing a dual-plane network with a single-plane network with link failures, the dual-plane network would have twice the total number of failed links. In our current results, it is possible to make this comparison by comparing data points in Figures 4.5, 4.6, 4.7, and 4.8. With that comparison, although arguably within a limited range, our findings show that application performance is less affected by link failure in multi-rail-multi-planar networks.

In later chapters we will build on our work here to conduct an in-depth study of how congestion can arise in a network and develop methods for mitigation of this congestion. Link failures are a cause of congestion in networks; and, unless these failures are dealt with, more congestion will ensue.

4.9 Conclusion

In this chapter, we presented the challenges of routing packets in Dragonfly-class interconnection networks with link failures. Specifically, when link failures are present, additional care must be taken when routing packets in order to avoid problems such as head-of-line blocking or deadlock. To utilize the network models in the CODES simulator with no modification of virtual channel assignment, we needed in order to dynamically find paths

that did not violate the assignment constraints. We developed an organizational structure in CODES that uses Algorithm 1 to accomplish that task. Specifically, the algorithm enumerates valid paths used by the progressive adaptive routing algorithms implemented in the CODES Dragonfly-class network models so that these algorithms are failure-aware.

We also extended the 1D-Dragonfly and Megafly CODES network models to support multi-rail-multi-planar configurations. We studied how single- and dual-planar Dragonfly-class networks handle increased levels of link failure using dynamic failure-aware adaptive routing.

We showed how simulation could be used to generate predictions of the behavior of irregular networks, such as those with failed links. We found that with a fixed number of failed links, dual-planar networks maintain a distinct advantage against communication failure and tolerance of link-failure-caused congestion over single-planar configurations, even with reduced bandwidth.

PART II

CONGESTION CONTROL

CHAPTER 5
DESIGN, IMPLEMENTATION, AND EVALUATION OF A CENTRALIZED SYSTEM FOR BROAD-SCALE CONGESTION MANAGEMENT

Advances in high-performance computing networks seek to maximize the performance of the available interconnect. Given a set limitation on router radix, the addition of more links to reduce overall network load may not be an option. With more and more applications vying for network resources, resource contention can become increasingly common.

This contention can become even worse when aggressive applications inject more traffic than the network can effectively handle resulting in congestion. We seek to find a mechanism to detect, identify the cause, and treat this congestion should it manifest in a given network. This mechanism will be implemented and tested on a simulated 1D Dragonfly network and discussed at length in this Chapter.

5.1 Introduction

Each year, newer supercomputing system designs leverage the latest advances in processor, memory, and storage technology to continuously push the limits in performance and power efficiency. Blindly applying faster components will not necessarily yield optimal performance results, however. Any communication between application processes distributed across a High-Performance Computing system must traverse a communication interconnect.

Analogously, the interconnection network is the circulatory system of the supercomputer. It carries packets of information between processes allowing them to work together to achieve a global problem goal. If, however, transient en-route packets encounter congestion on their path, then the arrival time of those packets will be negatively impacted. Ultimately, packets encountering congestion and being delayed is the exact mechanism for which packet interference - the incidence where packets from two different jobs find themselves in the same router buffer competing for resources - to occur. If this happens frequently enough, application performance can suffer as a result - hurting latency-sensitive applications the most.

Portions of this chapter have been submitted to: N. McGlohon *et al.*, "Exploration of congestion control techniques on dragonfly-class HPC networks through simulation," in *2021 IEEE Int. Conf. Cluster Comput.*

To make matters worse, once network congestion manifests, it can spread. Like cars avoiding a large backup on a highway by exiting onto side streets and, as a result, causing induced congestion on the very routes taken to avoid the initial congestion, when packets adaptively route around known congestion, this may cause more congestion hot spots to appear in the network.

Upon the formation of a congestion hot spot, if all trends in the network are maintained, and nothing is done to actively treat the congestion, there is little long-term hope at the situation resolving on its own. The best way to combat congestion is to take measures to avoid it in the first place. It is possible that reducing the overall network allocation, temporally spreading out the execution of applications could result in improved performance and reduced interference. This comes at the cost of slower total application set runtime but each application observes significantly lower latency during its execution.

Congestion cannot always be avoided. Some traffic patterns have the tendency to overload the processing capabilities of a singular or set of routers. Incast, or a many-to-one pattern, is a type of traffic that is likely to cause congestion in the network. In this type of traffic, there is often a request that is broadcast to participating application ranks who, in response, send packets back to the originating node. The result is a large number of packets originating from various places on the network funneled into a single router connected to the requesting node. Any other traffic that might be attempting to traverse this congested router will also be impacted.

To combat congestion once it has already been discovered on the network, a process called congestion abatement can be used to treat it. The process of congestion abatement is rather simple. Imagine a funnel placed under the running faucet in a kitchen sink; if the funnel's neck is not wide enough to expel water from the funnel at least as fast as water is being poured into it from the faucet, then the funnel will become congested and will eventually overflow. The obvious solution (aside from buying a larger and wider funnel) is to simply reduce the flow from the faucet. Congestion abatement works in a similar manner: it reduces the injection rate of nodes contributing to the congestion in the network. This process is maintained until the congestion is resolved.

This chapter details the design, simulated implementation, and evaluation of a congestion management system that will detect, identify cause, and treat congestion once it occurs in the network. This work was completed by extending the CODES interconnect network simulation

framework and is designed to be flexible and easily extendable to support different strategies and metric collection methods. The general design for this system is derived from publicly available patents submitted by Cray, inc. for the Aries router network systems [67]–[69]. That being said, it was written to be flexible and extensible for simulating different potential designs with varying degrees of data locality.

5.1.1 Three-Tined Resolution Approach

Effectively treating congestion without negatively impacting unrelated applications requires three mechanisms:

1. A method for detecting congestion in the network
2. A method for identifying the causal job(s) of the congestion
3. A method for abating the congestion, directed at said job(s)

These three mechanisms can, steady-state, be referred to as **Congestion Detection**, **Congestion Causation**, **Congestion Abatement** and will be referred to as such for the remainder of this chapter. Without any one of these modules, battling congestion effectively may prove to be rather difficult.

5.1.1.1 *Congestion Detection*

In order to work to reduce congestion in the network once it exists, it must be possible to detect the congestion in-situ. This will, generally, require some amount of information to be communicated to a congestion management layer. Metrics such as the number of stalled packets on a particular port, the number of injected and ejected packets in the network, and the average amount of time packets spend in port buffers can be used to determine if congestion exists in the network. An un-congested network will have fewer stalled packets with short buffer wait times.

It is also important to recognize the difference between short, instantaneous moments of congestion and long, sustained, steady congestion. The former will likely have little effect on application performance or will be too short to effectively treat. The system is designed to only detect congestion if certain criteria are met that indicates congestion over a continued period of time.

The congestion detection module essentially tries to find the solution to the following decision problem: *Is the network in a state of steady, sustained congestion?*

5.1.1.2 *Congestion Causation*

After the detection of congestion in the network, the next steps are to treat it by way of congestion abatement. Blindly applying abatement mechanisms to any or all injecting nodes in the network has a strong possibility of negatively impacting jobs that were unrelated to the congestion - jobs that were well-behaved or only communicating using routers on a different part of the network than where the congestion exists.

Identifying the job, or jobs, responsible for creating the congestion is necessary for precisely treating the congestion at its source. A job that is identified as the culprit is referred to as an **aggressor** job. Aggressor jobs are those that are targeted by the abatement mechanism should it be active.

The congestion causation module essentially tries to answer the following classification problem for each job in the network and the ranks that form it: *Is this job well-behaved, or is it an aggressor?*

5.1.1.3 *Congestion Abatement*

Finally, after congestion is detected and the causal job and ranks are identified, it is time for the congestion abatement module to activate. This module sends a signal to all aggressor job ranks identified by the congestion causation module to slow down their rate of packet injection. If the identified jobs are indeed the cause of the sustained congestion, then limiting their ability to inject traffic into the network should allow the network to naturally recover to a more normal state of operation.

If the congestion is sufficiently resolved, then the abatement mechanism can be lifted, and normal packet injection can be enjoyed by all network ranks. If congestion manifests a second time, then the process starts over with the abatement module slowing injection of the then causal, aggressor, ranks. There is little computation or logic to this module; it is simply the method of action utilized by the congestion management platform in conjunction with the other two modules.

5.2 Design

The proposed and implemented system design is based on publicly available information from U.S. Patents authored by Cray, inc. [67]–[69]. These patents describe a similar three-tined approach to congestion management for use on the Aries router-based interconnects. Newer router and system designs such as the Rosetta router and the Slingshot interconnect claim to use a newer congestion management design, but specifics regarding it are largely unknown to the public.

The congestion management system described, which will be referred to in this chapter as CODES Congestion Control (***CODES-CC***), largely takes influence from the admittedly non-specific information found in the related patents. The patents describe a hierarchical system of congestion controllers. These controllers described in the patent could be found alongside NICs, within routers, within cabinets, within a group of cabinets and so on; responsibilities are divided amongst them based on their rank in the respective hierarchy.

CODES-CC abstracts the congestion management communication layer. It assumes that the transmission of congestion information to the necessary actors is a solved problem. The intent of this work is to judge the capabilities of a theoretical congestion management solution, and fully simulating a secondary network for carrying managerial information is unnecessarily complex and may muddy the waters when determining the efficacy of the proposed solution.

The general flow of this congestion management design can be seen in Figure 5.1.

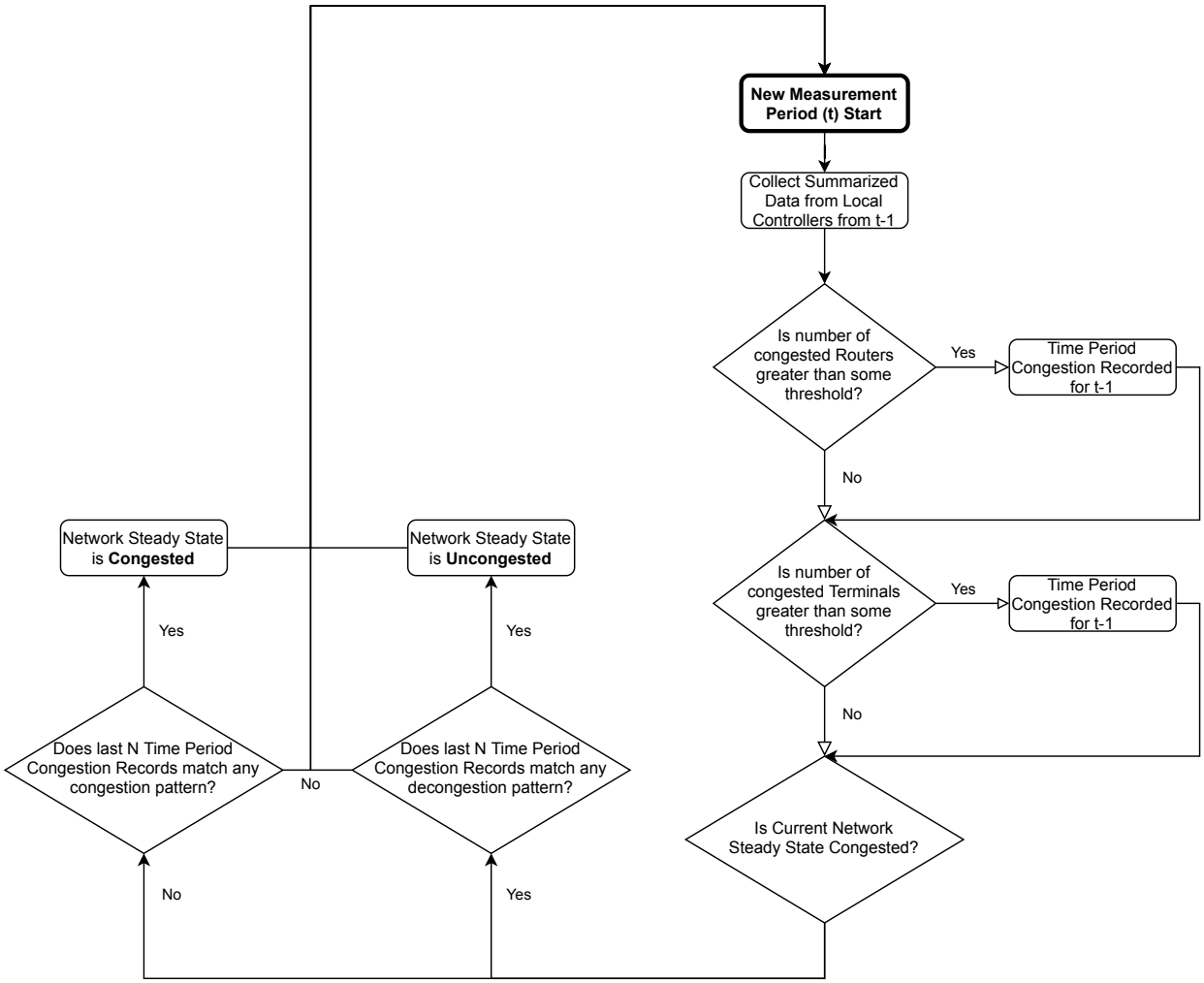


Figure 5.1: Flow chart describing the behavior of the main congestion control system loop from the perspective of the supervisory controller.

In Figure 5.1, we start with a new measurement period at the top. The system collects summarized data from the local controllers collected from the last measurement period. The system determines whether there was congestion observed in that measurement period according to criteria on either the routers or the terminals. From here, depending on whether the system is currently experiencing known steady congestion, it will see if the history of the last N measurement periods match any of a pre-specified set of congestion or decongestion patterns, where N is a configurable number of measurement periods to analyze for a matching pattern. If a matching pattern for congestion (if network is currently not congested) or decongestion (if the network is currently congested), then the steady-state of the network becomes either congested or decongested, respectively, and the system loop continues.

5.2.1 Responsibility Hierarchy

Similar to the cited patents, ***CODES-CC*** relies on a hierarchical structure to divide responsibilities amongst congestion control agents.

In the implemented simulation, each router and compute node terminal has an onboard **Local Controller** that acts as the liaison for ***CODES-CC***. It collects metrics related to congestion at that router/node and sends and receives messages from the ***CODES-CC*** system.

Again, in order to keep simulation complexity to a minimum, there is a singleton **Supervisory Controller** that serves as the collector of summarized congestion information and performs the managerial tasks of congestion detection and causation determination. It also makes decisions on whether or not to turn on congestion abatement and will send abatement-related signals to local controllers on specified compute node terminals. The responsibilities of this singleton supervisory controller could be distributed, which may result in added simulation realism as well as some parallel performance efficiency improvements. Said distributed design is explored in Chapter 6.

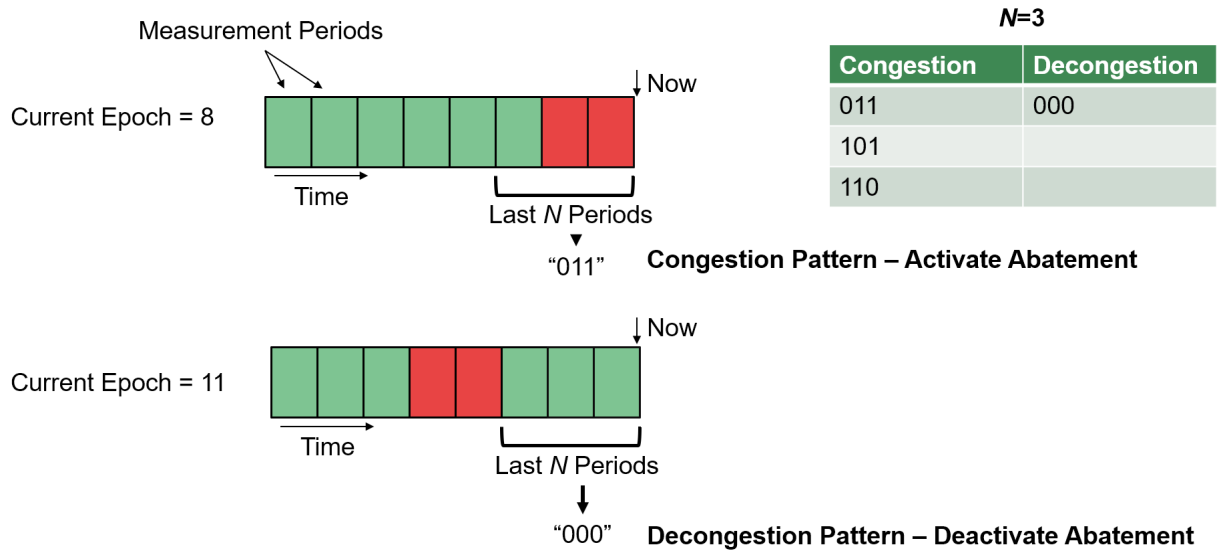
5.2.2 Decision Making

There is a core system loop in ***CODES-CC*** that drives the congestion control mechanisms and monitors the simulated network performance. During a given CODES network simulation, measurement periods are defined. This ultimately defines the granularity of ***CODES-CC***. At the end of each measurement period, local controllers collect performance information collected during said period, summarize it, and send it to the supervisory controller.

The supervisory controller, after receiving information summaries for the recently passed measurement period, will activate the decision-making mechanism of the Congestion Detection module. This will determine two things:

1. Did the metrics from the last measurement period represent a network state of congestion?

2. Did the last N measurement periods match a pre-defined pattern of congestion or de-congestion?



N=3

Congestion	Decongestion
011	000
101	
110	

Figure 5.2: An example of how measurement period history is represented as a string for matching against pre-defined patterns of congestion and decongestion.

.

The first determines, specifically, if the last measurement period meets pre-specified criteria for *instantaneous* congestion. The collection of these instantaneous congestion statuses will be used to determine whether the network is experiencing *steady, sustained* congestion. If a single measurement period is congested but is resolved by the next measurement period, then activating abatement could be heavy-handed and might needlessly impact application performance.

After determining the instantaneous state of congestion for the last measurement period, the supervisory controller looks at a configurable length of measurement period history. If the network is currently not steady-state congested, then it will compare the measurement period history with a set of pre-defined congestion patterns. These patterns are a string of 0s and 1s representing the instantaneous congestion state of the last N measurement periods. If there is a match in this set of patterns, then the network will be considered steady-state congested. Conversely, if the network is currently stead-state congested, then it will compare the measurement period history with a set of pre-defined de-congestion patterns to find a match before declaring the network congestion resolved. A visualization of how these measurement period patterns correspond to recognized congestion and decongestion criteria is shown in Figure 5.2.

Local controllers do not make decisions regarding congestion management; they simply report information to the supervisory controller, who, in turn, makes decisions and sends congestion management commands down to respective local controllers.

5.2.3 Aggressor Identification

Once congestion has been detected, in order to effectively treat it, it is important to determine what ranks from what job are responsible. Identifying the problem (aggressor) job can be considered a problem of classification. Given a set of jobs and metrics associated with their network usage, the task of this module is to classify jobs as aggressors or non-aggressors.

Ultimately, given a single metric that can separate high relative network usage jobs from low relative network usage jobs, it should be possible to separate the set of jobs into the two classifications. One potential metric can be determined by collecting packet injection and ejection counts for each job and derive the total number of packets that are currently in-flight in the network. Normalizing this number by the total number of ranks in the respective job will yield a value that grows as the total network usage per rank of a job increases.

This normalization step is to prevent the possible case where there is a single large job that allocates half of the available nodes in the network but is otherwise well-behaved. While it may be contributing to the overall network usage, it is expected to due to its larger size. Smaller jobs that are injecting disproportionately more traffic than what might be expected from so few ranks are a source of imbalance in the network. They may be taking a greater share of the network resources than what might be expected of a job allocated to so few compute nodes.

Aggressor identification in the Slingshot interconnect appears to take an even finer-grained approach. It does not attempt to find aggressor jobs, but aggressor ranks instead. A job with 1000 ranks may have 100 or so that are misbehaving and causing increased network load. To classify and abate the entire job as an aggressor may unnecessarily harm the performance of the 900 other well-behaved ranks. How exactly this is achieved is not yet publicly known.

5.2.4 Congestion Abatement Policy

The method of action for the treatment side of congestion management comes from the abatement policy. The idea behind congestion abatement is to take an active approach

toward reducing the overall network load by reducing the rate at which packets are injected into the network. If – through detection and aggressor identification – this policy can be targeted to specific ranks, then the detrimental impact on well-behaved jobs and ranks will be minimized.

Congestion abatement is activated after congestion has been first detected and an aggressor has been identified. Then the supervisory controller sends an abatement signal to the ranks of the aggressor application to abate according to the implemented abatement policy.

There are two designs for this policy that have been sketched out during the course of this work. The simplest design is a flat bandwidth penalty that is applied to abated ranks. Any packet injected from affected ranks will be done with reduced bandwidth. The simplicity reduces the configuration complexity, which is helpful considering that ***CODES-CC*** has a substantial degree of configurability to being with. That being said, new schemes can be devised and implemented should they be desired.

Another potential abatement design takes from recent work on Quality of Service (QoS) techniques in CODES. Specifically, that work increased the flexibility and realism of the QoS implementation to utilize token bucket rate-limiting, which makes the system more capable of correctly handling 'bursty' traffic patterns. A packet can be scheduled for output if and only if a token exists in a bucket. There is a pre-defined rate at which tokens are filled into the bucket, and so if there are long periods of no packets being scheduled on a given port, the bucket can fill up enough so that a burst of packets can be scheduled even if the desired rate exceeds that of the bucket filling rate.

One potential way to utilize token bucket rate-limiting for congestion abatement would be to establish two buckets per point of injection. One bucket will fill very slowly with some guaranteed minimum rate of injection. Another bucket will have tokens that are only generated when a packet sent from a specific rank has been confirmed to be ejected from the network. The first bucket will prevent starvation, allowing at least some baseline level of packet injection while the second bucket will try to cap the number of packets that any one rank can inject into the network. More work will need to be done to find good configuration schemes before this is implemented.

5.3 Implementation

The work discussed in this chapter is all implemented in C/C++ on the CODES interconnection network simulation platform. Any presented results are from simulations performed using code available in the CODES online GitHub git repository. CODES has been used in a large number of academic publications focusing on topics from packet routing to prototypical network topologies.

5.3.1 General and Global Information

There are a few things added to the core operation of CODES in order to make the ***CODES-CC*** system integrate and work properly. Most notably, there is a globally visible variable, `g_congestion_control_enabled` that is 1 if the ***CODES-CC*** system is enabled; otherwise, it is 0. When CODES is initialized, it loads in a configuration file that tells it how many of what type of LP must be instantiated. For ***CODES-CC***, there had to be a single additional LP mapped into the simulation representing the supervisory controller. Since CODES typically expects all LPs to be defined in a repeated pattern format, it was not as simple as extending the CODES configuration file to include the supervisory controller. The CODES mapping initialization function was extended to define a single LP after all others had been defined normally.

Because the LP global identifier (GID) can easily change depending on how the simulation is configured at runtime, the GID for the supervisory controller, once it is determined upon simulation initialization, is stored as a global variable: `g_cc_supervisory_controller_gid`. This value is used by all agents in the ***CODES-CC*** system any time a message must be sent to the supervisory controller.

The ***CODES-CC*** system utilizes its own ROSS message types for communicating information between congestion control actors (the supervisory and local controllers). These message types are as follows:

`CC_SC_HEARTBEAT` This event is continuously sent by the supervisory controller with a delay of the length of the measurement period so that it will 'wake up' and process its next step in the ***CODES-CC*** system loop. If the supervisory controller is aware that every workload in the simulation has completed, it will stop sending this heartbeat message to itself.

`CC_SC_SIGNAL_ABATE` This event is sent from the supervisory controller to the local controllers of identified aggressor jobs on respective terminal LPs should congestion be detected by the system. Upon receipt of this event, the local controller on the terminal LP will adjust its injection bandwidth according to the strategy employed by the ***CODES-CC*** system. If, through a possible out-of-order event, an abatement signal is received while the local controller is already operating with reduced bandwidth, nothing further happens.

`CC_SC_SIGNAL_NORMAL` Like the previous abatement signal, this event is sent from the supervisory controller to the local controllers of the previously identified aggressor job ranks. Upon receipt of this event, the local controller on the terminal LP will reset its injection bandwidth to normal as the abatement order has been lifted.

`CC_RLC_HEARTBEAT` This event is continuously sent by the local controller on a router LP with a delay of the length of the measurement period so that it will 'wake up' and report information gathered over the course of the last measurement period to the supervisory controller. If the local controller has knowledge that all workloads in the simulation have completed, the heartbeat message will not be sent.

`CC_R_PERF_REPORT` This event is sent from the local controller on the router LP to the supervisory controller. Within this event is encoded the performance metrics summarized by the local controller on the router. In the current implementation, this is the number of ports on each router that have been listed as stalled.

`CC_TLC_HEARTBEAT` This event is continuously sent by the local controller on the compute node terminal with a delay of the length of the measurement period so that it will also wake up similarly to the router LP local controller and report its information to the supervisory controller. It will also verify that all workloads in the simulation have not yet been completed before sending another heartbeat message.

`CC_N_PERF_REPORT` This event is sent from the local controller on the terminal LP to the supervisory controller. In its current implementation, this message contains whether or not the port of injection on the compute node is stalled or not.

`CC_WORKLOAD_RANK_COMPLETE` This event is sent by numerous LPs in the simulation. Its primary use is to notify recipients of any or all workload completions depending on the context. Workload LPs that generate traffic to be injected by the compute nodes send this event to the supervisory controller to notify it that the workload rank indicated has completed. The supervisory controller keeps a count of all workload ranks in the simulation and their completion status. Once the supervisory controller has received a workload complete message from all jobs in the simulation, it will send a `CC_WORKLOAD_RANK_COMPLETE` message to all router and terminal LPs to inform them that they no longer need to send heartbeat messages to themselves for congestion control. If any congestion control actors do not stop sending their heartbeat messages, the simulation will never end.

5.3.2 Supervisory Controller

The supervisory controller LP manages the state of the ***CODES-CC*** system. It receives information from local controllers, makes decisions, and sends abatement signals back down to local controllers.

5.3.2.1 *LP State*

It keeps track of the following state based on simulation configuration, information received from local controllers, and decisions made during its behavior loop:

1. A map of router IDs to the number of stalled ports registered during the last measurement period
2. A map of terminal IDs to the stalled status of the respective terminal during the last measurement period
3. A full history map of measurement period serial number to whether the criteria for instantaneous congestion was met for ports in the system
4. A full history map of measurement period serial number to whether the criteria for instantaneous congestion was met for the terminals in the system
5. A map of terminal ID to the total injected packet count as of the last measurement period's end

6. A map of terminal ID to the total ejected packet count as of the last measurement period's end

7. A map of application ID to the total number of in-transit packets as of the last measurement period's end

8. Maps of application ID to sets of terminal IDs and LP GIDs associated with said application

9. Record of the currently abated application, if there is one

10. Record of the current measurement period/epoch and how many have had instantaneous congestion

11. Boolean values for whether there is instantaneous router/terminal congestion, whether abatement is active, and whether all workloads have completed

5.3.2.2 *LP Behavior*

The behavior loop of the supervisory controller is driven by its heartbeat messages. Upon receipt of a heartbeat message, the supervisory controller will run through each module of the congestion control system in this order: Congestion Detection, Congestion Causation, Congestion Abatement.

Congestion Detection Upon starting the congestion detection behavior, the supervisory controller will seek to determine if the network, over the last measurement period, experienced instantaneous congestion. There are two ways that the network can experience this instantaneous congestion: from the terminal injection ports and from the router ports. If either of those is congested for that measurement period, then that measurement period is set as congested (boolean `OR`).

In determining whether there is widespread congestion from the terminal injection ports, the total number of stalled injection ports is collected from the terminal local controllers' performance reports. If the sum of all stalled injection ports crosses some configured percentage threshold, then the network terminals are experiencing congestion. Similarly, the total number of stalled router ports is collected from the router local controllers' performance

reports. If the sum of all stalled router ports crosses some percentage threshold, then the network routers are experiencing congestion.

After determining the congestion status for the last measurement period, the steady-state congestion detection feature becomes active. This looks at the last N measurement periods and compares them to a pre-defined set of congestion or decongestion patterns. These patterns are loaded in from a human-readable text file at simulation initialization. Each line in the file represents a pattern for congestion or decongestion (depending on which file is being read).

If the network state is currently set to denote that it *is not* experiencing steady congestion, then it will look at the set of loaded congestion patterns. Conversely, if the network state is currently set to denote that it *is* experiencing steady congestion, then it will look at the set of loaded decongestion patterns. If and only if it matches one of those decongestion patterns will the status of steady congestion be lifted - even if the last N measurements do not match any congestion patterns.

Congestion Causation After determining whether or not there was congestion via the congestion detection module, the congestion causation module is activated. This analyzes the total number of in-transit packets on a per-application basis and normalizes this count based on the number of ranks that contribute to that application's traffic. The current implementation finds the job with the highest in-transit packet to rank ratio and considers that job the aggressor. If ranks are well separable with this metric (and any others in the future), this could be easily extended to support blaming multiple applications.

Congestion Abatement If the network congestion status changed during the activation of the congestion detection module, then the congestion abatement module will notify the necessary terminal local controllers of that change. For instance, if the congestion detection module registered that there was new steady congestion, then the congestion abatement module will take the input from the congestion causation module to determine what application's terminal local controllers to send an abatement signal to. If this congestion is maintained through the next measurement period, no additional abatement signal is sent. The only signal that should follow an abatement signal is a normal signal which is only sent if the network goes from a state of steady congestion to steady decongestion (the last N

measurement periods matched a decongestion pattern).

New Epoch After the three congestion modules have done their tasks, then the supervisory controller registers a new epoch and sends a new heartbeat message to itself as long as it knows that there exists at least one running workload left in the simulation. Otherwise, it completes its contribution to the simulation.

5.3.3 Router/Terminal Local Controller

All behavior for congestion control on the part of a router or terminal is performed by the existing router LP that represents it. This includes the forward and reverse handling of all congestion control-related state changes and events. CODES has been extended to support the facilitation of congestion control messages between the supervisory controller LP and router and terminal LPs.

5.3.3.1 *State*

The local controllers keep track of the total packets routed or injected per port in the last measurement period, as well as how many packets were stalled per port in the last measurement period. The local controllers use these metrics to determine whether ports or the terminal have stalled. In the current implementation, if there are more packets stalled than successfully routed during a measurement period, that port or terminal is considered to have been stalled. It also keeps track of whether all workloads have completed in the simulation.

5.3.3.2 *Behavior*

The local controller behavior is pretty simple. With every heartbeat event that it receives, it determines how many of its ports or injection rails have stalled, sends that count to the supervisory controller, registers a new measurement period, and sends another heartbeat message to itself. If the local controller is aware that all workloads in the simulation have completed, then it will not continue to send heartbeat messages to itself.

5.4 Evaluation

The majority of this work has been in the development of the platform as an easily extendable and configurable testbed for various approaches to congestion management that meet the communication paradigm discussed in the section on design - notably, with routers and terminals periodically sending congestion information to a global entity (in current implementation: a singleton Supervisory Controller).

This grants the supervisory controller periodic summarized global information about the network, making it somewhat of an oracle. If the supervisory controller, as an oracle with global information, cannot utilize its collected information to the network's benefit, then it is likely that any introduction of error or delays from a more realistic or distributed solution will also not be of benefit. Since the goal of this work is to understand what strategies could be utilized to mitigate and treat congestion, the experiments used to evaluate the strategies for congestion detection and abatement utilize this singleton supervisory controller implementation despite its unlikely use in real-world systems.

5.4.1 Experiments

The platform has been implemented and tested on the CODES 1D Dragonfly network model. The specific network configuration utilized to test ***CODES-CC*** is a Dragonfly network with 3,078 compute nodes. A couple of workload sets were applied to the network with and without congestion management to observe the effect of the ***CODES-CC*** system on workload application performance. To prevent potential bias from accidental 'perfect' job placement, the mapping of job rank to terminal is entirely random for each workload.

5.4.1.1 *Traces and Synthetic Aggressor*

The workload sets described in this subsection utilize DUMPI application traces. These are files that contain information regarding communication patterns utilized by the DUMPI represented application. This yields communication patterns that are more realistic than synthetic procedurally generated traffic.

The two application traces featured are the AMG1728 and MG1000 DUMPI traces. These represent two different multi-grid solver applications and have been utilized in the evaluation of network performance in previous works. Two workload sets of this type were tested, with and without ***CODES-CC***. The first set was AMG1728+MG1000, and the

second set was AMG1728+MG1000+SYN350. SYN350 is a uniform random synthetic traffic pattern that injects a substantial amount of traffic into the network to attempt to interfere with the communication performance of the two DUMPI trace workloads.

5.4.1.2 *Intel SWMs*

The workload sets described in this subsection utilize the Intel Scalable Workload Models that procedurally emulate the behaviors of real-world applications. The first workload set is LAMMPS2048+SYN1000, where the synthetic uniform random traffic is supplying a relatively low level of background traffic just to simulate general use on the network. The second workload set is LAMMPS2048+SYN1000+INCAST30, where the incast workload will periodically perform an incast traffic pattern of 29 ranks to 1.

5.4.2 Observations

The largest challenge with gathering results from this system is the large input space: there are many different parameters to configure and it is difficult to optimize. That being said, it was generally possible to find some configuration that performed better with the system enabled when compared to network performance without ***CODES-CC***.

5.4.2.1 *Trace-Based Experiment*

Taking a look at the trace-based experiment, trace application performance took an impact with the addition of the synthetic aggressor traffic. If the ***CODES-CC*** system was enabled, then the congestion was detected, the synthetic job was recognized and identified as the aggressor, and its ranks were abated. The performance impact on the trace applications is still worse compared to when the aggressor traffic did not exist but is notably improved with ***CODES-CC*** turned on.

Another benefit of a congestion control system is the overall reduction in packet latencies across all applications in the network.

5.4.2.2 *SWM Incast Based Experiment*

It has been noted that Incast is a particularly interesting traffic pattern as it is a common pattern used in many different applications. It is also something that can, by itself, cause significant sudden traffic across the network and focusing down to a single destination router. That destination router can easily become overwhelmed by the number of packets

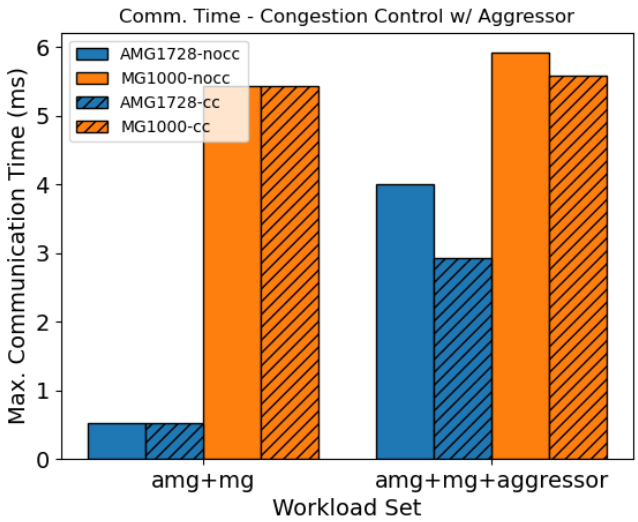
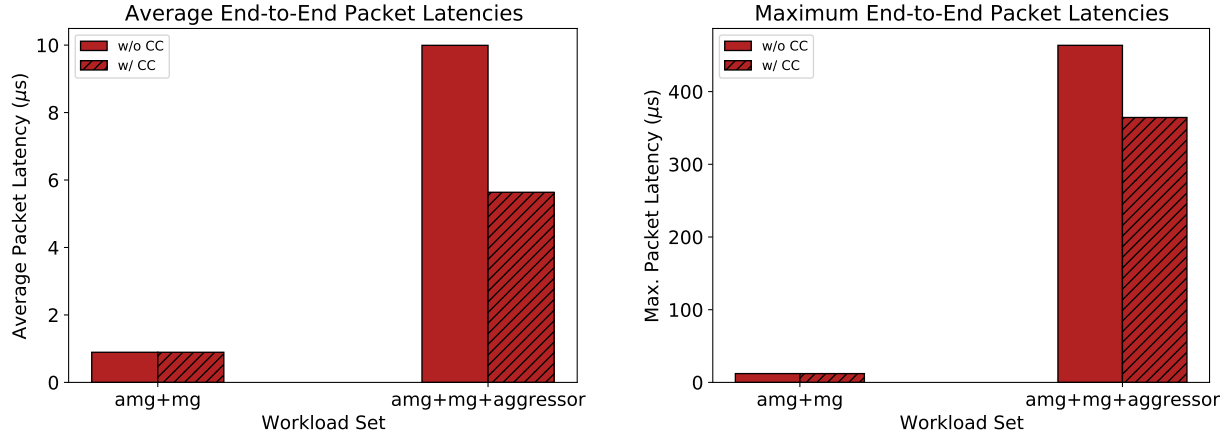


Figure 5.3: Comparing two workload sets, one without and one with a synthetic aggressor job. Shaded bars represent application performance with congestion control enabled.



(a) (b)

Figure 5.4: Comparing the mean (a) and maximum (b) end-to-end packet latencies with and without *CODES-CC* enabled. *CODES-CC* made minimal to no impact on the workload set without the synthetic aggressor but significantly reduced latencies in contrast to when an aggressor workload is introduced.

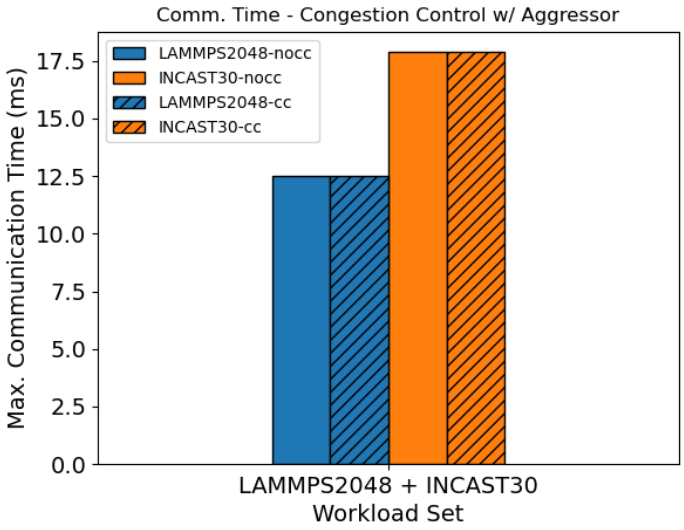


Figure 5.5: Performance of the LAMMPS2048 SWM in conjunction with the INCAST30 SWM. Shaded bars represent application performance with *CODES-CC* enabled. In this case, while it is noted that *CODES-CC* was able to identify congestion and attribute it to the INCAST30 SWM, it was not capable of abating it.

suddenly routed to it, and any other in-route packets that happen to be directed through said overwhelmed router may experience interference.

As observed in Figure 5.5, while the ***CODES-CC*** system was able to detect the congestion caused by the Incast workload and correctly attribute it as the aggressor, at that point it is too late to abate, the packets that are causing the congestion have already been injected, and the abatement mechanism could not adequately prevent interference and made no impact on packet latencies.

5.4.2.3 *Improving Congestion Control and Comparing with Quality of Service*

With the observation from the last Incast experiment, one could imagine a policy where an aggressor, after being identified as such, is permanently penalized and its traffic is abated. This would prevent the case seen where the Incast workload can subvert the effects of abatement entirely.

Working toward this idea, we set up a new set of simulations. A larger incast workload was used to create a more noticeable interference impact on the primary workload. Additionally, once congestion is detected and the aggressor job is identified, abatement is activated and remains active until the aggressor job is the only workload left on the network.

It is also worth comparing the performance of ***CODES-CC*** in comparison with the previously implemented CODES Quality of Service (QoS) techniques.

Figure 5.6 shows that ensuring that the abated workload is permanently penalized for causing congestion will result in improved performance of the primary workload. In comparison with QoS techniques, however, it was unable to restore baseline performance. One contributing factor to this result is that the abatement mechanism only modifies the speed at which packets are introduced into the network; once packets are in the network, nothing else is done to prevent resource contention.

This led to the notion that perhaps the abatement strength is not of large enough magnitude to prevent the aggressor incast workload from creating a hostile situation near the point of packet convergence (the target of the incast). This makes sense as the problem is not just a large number of packets in one localized area, but that the large number of packets are "bottlenecked" by the rate of packet ejection of the router. This means that in order to prevent the incast workload from significantly interfering with the traffic from the primary workload, the rate of injection must be reduced by a factor proportional to the size of the incast itself.

In this next result, we strengthen the magnitude of the abatement mechanism so that the allowed usable injection bandwidth of the aggressor workload is the base injection bandwidth multiplied by 1/(number of incast ranks), or 0.01. Doing this significantly reduced the impact of the incast workload on the primary with minimal impact on the incast workload's overall performance. This negligible impact demonstrates that the bottleneck of the incast workload previously was not due to its rate of injection but instead due to the amount of time that it must wait for all packets of one incast phase to complete before starting another.

Figure 5.7a shows that increasing the strength of the abatement protocol yields a significant improvement in the performance of ***CODES-CC***. However, it is still not strong enough to restore baseline performance. One contributor to this is that these experiments utilized a random job-to-terminal mapping, so the jobs are forced to share links. As such, without ensuring prioritized lanes, traffic will inevitably be fighting for network resources and result in some level of interference.

Routing is also something to consider. Until now, all of the experiments evaluating ***CODES-CC*** have utilized the Progressive Adaptive Routing feature of the 1D Dragonfly network model. However, what adaptive routing does is spread congestion across the network

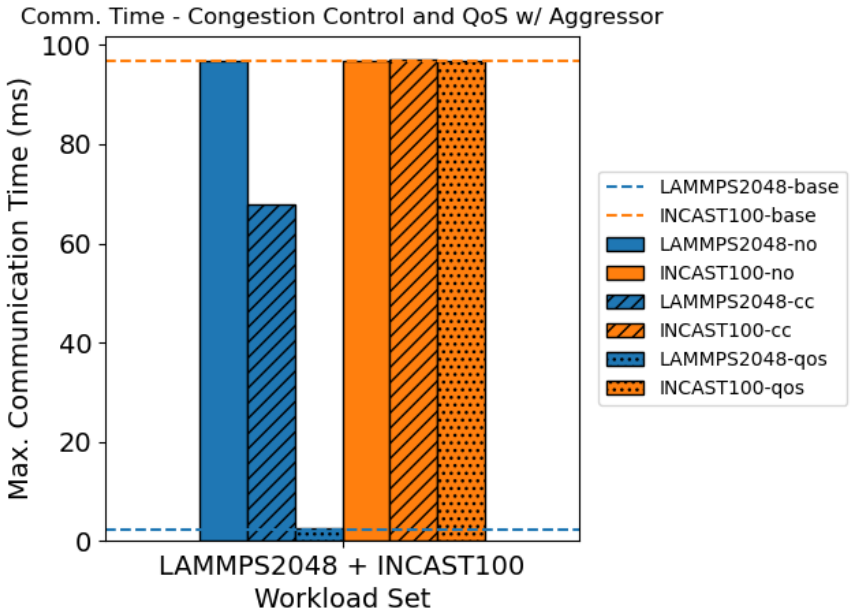
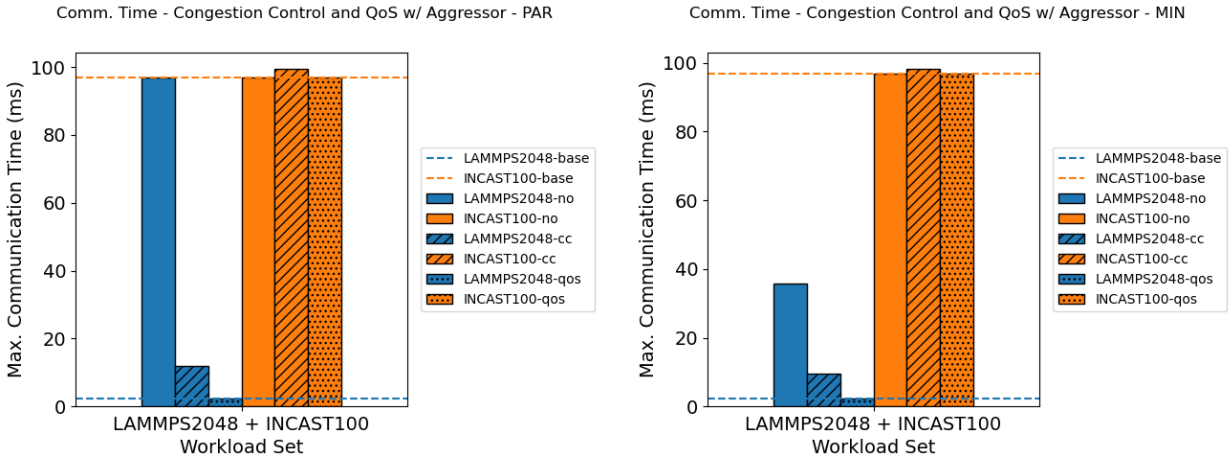


Figure 5.6: Performance of the LAMMPS2048 SWM in conjunction with the larger INCAST100 SWM without any control features, with *CODES-CC*, and with QoS. In this case, when an aggressor is identified, it is permanently abated.



(a) (b)

Figure 5.7: Performance of the LAMMPS2048 SWM in conjunction with the larger INCAST100 SWM without any control features, with *CODES-CC*, and with QoS for progressive adaptive (a) and minimal (b) routing. Also pictured as a dotted line is the performance of the respective workload when completely isolated on the network with no control features enabled. In this case, when *CODES-CC* is active, the strength of abatement is proportional to the number of ranks in the aggressor job (x0.01 bandwidth multiplier).

to more effectively balance the load across links of the network. This can sometimes cause competition for resources in areas of the network that would have otherwise been fairly isolated as non-minimal tagged packets take an intermediate route to a random network destination before being routed to its final destination. Figure 5.7b shows that enabling minimal routing does improve the un-controlled traffic case but does not greatly change the baseline or ***CODES-CC***/QoS performance. This does imply, however, that a different allocation mapping might improve results in general and make things easier for ***CODES-CC*** to improve.

5.4.2.4 *Runtime Impact*

Because the system relies on a periodic large amount of simulation messages from the performance reports resulting from every heartbeat message, there are a lot of additional simulation events to be processed by PEs. As one would expect when comparing two simulations, the one with more events to be processed, all other things being equal, will have longer runtime.

The platform has been tested on 1D Dragonfly networks with as few as 72 nodes and as many as 8,320 nodes with acceptable levels of runtime performance impact. The level of impact is directly correlated with the desired granularity of the ***CODES-CC*** system: the length of each measurement period. The more measurement periods there are in the span of the simulation, the more times routers and terminals report information to the supervisory controller.

In the trace-based experiments, there was about 5% additional runtime attributed to the overhead of ***CODES-CC*** and was ultimately negligible. This was, however, not necessarily the case when applying ***CODES-CC*** to workload sets that used the SWMs. The SWM models tested in this work, LAMMPS2048 and INCAST30, tended to have overall communication spanned over tens of milliseconds in simulation time despite being relatively lightweight in the intensity of communication and, therefore, simulated events. In these simple tests, there was an 800% increase in runtime when ***CODES-CC*** was enabled.

The current design of ***CODES-CC*** has a static time window for the measurement period where every router and terminal LP will send performance metrics. By using the granularity necessary to be able to detect congestion adequately, there is the possibility of the system generating a lot of events from ***CODES-CC*** overhead that do not actually convey

much information at all. This is exactly what is happening in the SWM experiments. The number of events coordinating the ***CODES-CC*** system vastly outnumber those representing network events. This should be a focus of optimization in the future.

5.4.3 Strengths and Weaknesses

The ***CODES-CC*** system has strengths and weaknesses. While not perfect, it does at least show that there is merit to the idea and that perhaps with proper tuning and metric choice, there could be significant performance boosts found from congestion abatement.

When the system worked, it was able to significantly reduce the average and maximum packet latencies across all applications in the network, including the one being abated. It is important to note that while there was some time spent trying to find a good system configuration to yield these results, it did not take particularly long to find one. With greater intuition gained from continued research, it would seem likely that better configurations could be found.

One observed weakness was the failure to affect the network congestion found from the Incast workload. The Incast workload will have bursty periods of all-to-one communication and will be in a period of low/no communication following. While not represented in any figures, the ***CODES-CC*** system was able to correctly detect congestion after the Incast workload entered a period of all-to-one communication. It also correctly identified the Incast workload as the aggressor. By the time the congestion was detected, however, the packets had already been injected, and the damage was done, the abatement policy would have no effect on the Incast workload as it would be in its period of no communication during the period that abatement was active. The congestion will, independent of the abatement and simply by nature of the workload's burstiness, resolve. Decongestion will be detected by ***CODES-CC*** and a normal signal will be sent to the Incast ranks. And the cycle will repeat until all workloads have completed, with the ***CODES-CC*** system never effectively doing anything to prevent or treat the congestion caused by the Incast workload.

If the ***CODES-CC*** policy is changed so that an abated job remains abated until completion (or it is the only job left on the network), then we are able to regain improvements with ***CODES-CC*** enabled as it bridges the gaps in burstiness of a workload like the Incast SWM.

Another weakness is the double-edged sword of flexibility. The system is very con-

figurable and flexible, but that configurability also means a large dimensional input space. Finding an optimum configuration parameter set may be a very difficult task. A step for the future would be to try and find ways that would allow the system to self-tune. Insight for how this might be possible may come from future public press releases of commercial systems that utilize congestion management systems (Slingshot).

5.5 Future Work

Through the course of development, testing, experimentation and observation, there are a number of areas for potential improvement. Firstly, there is the aforementioned new public information on how a modern congestion management system might approach things.

Secondly, the ***CODES-CC*** system was developed so that it could be expanded upon. More metrics and schemes can be implemented and utilized in conjunction with those already in use. Different metrics for determining if a port has stalled may prove to be more sensitive and useful, such as the amount of time that packets have spent 'on a router' waiting to be forwarded onward. It currently acts as a proof of concept that this technique is something worth pursuing, and a similarly angled mechanism (but with a different implementation/hierarchy) could yield significant performance benefits.

Thirdly and most importantly: the results, while promising, are not as significant as those found in previous work on Quality of Service techniques. QoS was able to, without network partitioning, benefit all applications on the network and reduce packet latencies and improve application performance across the board to a significant degree. And while there is not as much tunability and flexibility when compared with ***CODES-CC***, it did not really need all of that tunability in order to generate significantly improved performance. That being said, combining the two mechanisms to work together could be an interesting concept for future research. This could include using a form of congestion detection and causation to determine jobs that are causing congestion and relegate them to a lower priority traffic class as the mechanism of abatement.

5.6 Conclusion

Network congestion in an HPC system is a complicated problem. Most HPC interconnect platforms do not have the packet-dropping-and-retransmitting mechanism that ethernet does. As a result, once congestion manifests, it is a difficult problem to treat. Like cars on a

backed-up highway, eventually, all of the cars will need to reach their destination, and as long as more cars continue to "join the fray" and become part of the congestion, it will persist.

Congestion management solutions that utilize abatement mechanisms seek to reduce the continued packet influx to allow the network congestion to resolve itself. We have presented a system design and developed ***CODES-CC*** based on said design. ***CODES-CC*** is a simulated system for congestion control in CODES network models that detects congestion, identifies its causal application, and limits the continued packet injection rate of said aggressor application.

While this system has been shown to work as described, there is room for improvement and optimization. Additionally, there are cases shown where it has not proved useful. Knowing the strengths and weaknesses of the proposed design, however, is a promising step toward a universal method for preventing, diagnosing, and treating congestion in a high-performance computing communication network.

CHAPTER 6
DESIGN, IMPLEMENTATION, AND EVALUATION OF A DISTRIBUTED SYSTEM FOR PRECISE CONGESTION MANAGEMENT

Ensuring optimal communication latency in High Performance Computing (HPC) networks is of critical importance to the efficient operation of facilitated applications. Different application operations and types of tasks, such as IO operations, can create a variety of traffic patterns across the system interconnect. Some communication patterns, however, can be problematic for overall system performance.

One traffic pattern of particular concern is the many-to-one or incast. When packets sent from many different endpoints target a singular, or small number of destinations, it can overwhelm the receiving endpoints' ability to process the traffic, resulting in a cascading effect of induced congestion. This can have broad-reaching, detrimental effects to other applications as their data streams encounter induced congestion.

The concept of congestion control has been explored in various HPC system technologies and is an important feature in state-of-the-art networks such as Infiniband and the Cray Slingshot interconnect. Because access to physical, full-scale interconnects of bleeding-edge design can be challenging and the exact mechanisms of operation not publicly known, we look to simulation to explore techniques for congestion control with a fine level of flexibility not available on real-world systems.

We present and explore a mechanism for congestion control which seeks to detect network congestion, identify its cause, and abate it by throttling injection of identified aggressor endpoints. Our work proposes, discusses and evaluates two similar mechanisms for congestion control in two different network simulators and their capabilities at mitigating the effects of congestion on application communication performance and general system packet latencies.

Portions of this chapter have been submitted to: N. McGlohon *et al.*, "Exploration of congestion control techniques on dragonfly-class HPC networks through simulation," in *2021 IEEE Int. Conf. Cluster Comput.*

6.1 Introduction

The performance of applications on a High-Performance Computing (HPC) system is dependent on a number of factors. While specifications and design details play a significant role, one factor that is difficult to predict is the effect of application interference as a result of contention on the underlying communication network resources. As HPC innovation drives toward the threshold of 'exascale' computing [70], new technologies for managing system resource contention must be developed.

A single system may facilitate hundreds of independent jobs simultaneously, each vying for network resources. When network switches are overwhelmed with traffic, congestion can arise, leading to increased packet latency and decreased application communication performance. Making matters worse, once network congestion manifests, it can spread. Analogous to vehicles trying to avoid heavy congestion on a highway by taking alternate routes on side streets, secondary congestion will consequently form.

Once a hot spot forms, if nothing is done to actively treat the congestion, there is little hope of the situation resolving on its own. The best way to combat congestion is to take measures to avoid it in the first place. One solution would be reducing the overall network allocation, running fewer applications at a time. This might result in improved performance and reduced interference but comes at the cost of a lower total application throughput.

To combat congestion once it has already been discovered on the network, a process called congestion abatement can be used to treat it. The process of congestion abatement is rather simple: reduce the injection rate of nodes contributing to the congestion in the network. This process is maintained until the congestion is naturally resolved as the existing in-transit packets are routed and ejected from the network.

To effectively utilize congestion abatement, congestion must first be detected, and the causal ranks – the endpoints responsible for creating the problematic traffic pattern – identified. The three mechanisms: congestion detection, causation, and abatement have been implemented in CODES [62].

A major weakness of the congestion control system, ***CODES-CC***, described in Chapter 5, is its centralized approach. In a realistic scenario, this poses a scalability issue. Also, even under the best of circumstances, it required significant amounts of congestion to be recognized on the network before it could respond. And even when it could respond, it was challenging to identify which applications were at fault and how much their traffic should be

throttled to address the issue.

Furthermore, the previous design was very heavy-handed. When it did apply congestion abatement strategies, limiting the injection rate of selected endpoints, it would apply the strategies to *all* ranks of an offending job – not just the ones who were responsible for the congestion.

In order to estimate the capabilities of the latest developments in network architecture, we simulate networks with the same underlying topology as the HPE-Cray "Slingshot" interconnect [71], the 1D Dragonfly [13], using CODES. Workloads applied to these simulated networks include online workload models designed to emulate common real-world HPC applications and traffic patterns using the Scalable Workload Models (SWM) [32] library. We create problematic, aggressor, incast traffic patterns which overwhelm exiting port buffers and induce network congestion as a result.

In this chapter, we present simulated implementations for self-tuning rate-limiting congestion control in an HPC interconnect simulation platform dubbed ***CODES-CC2***. We explore the effects of congestion caused by problematic traffic patterns such as many-to-one, including the inadequacy of just using adaptive routing for handling this problem. Finally, we demonstrate the efficacy of the presented congestion control techniques at restoring network performance.

6.2 Background

We simulate an underlying HPC communication network topology known as the 1D Dragonfly. The interconnection scheme of Dragonfly networks features local *groups* of switches with some degree of connectivity between switches in each group. These groups of switches are also connected to each other with global links providing one or more connections between each pair of groups.

Dragonfly networks, dating back to the late 2000s, were originally designed as a highly scalable, cost-efficient innovation in HPC topology research. Notable real-world systems featuring a Dragonfly interconnect include the HPE-Cray "Cascade" XC series of supercomputers [14] and their latest "Slingshot" systems [71].

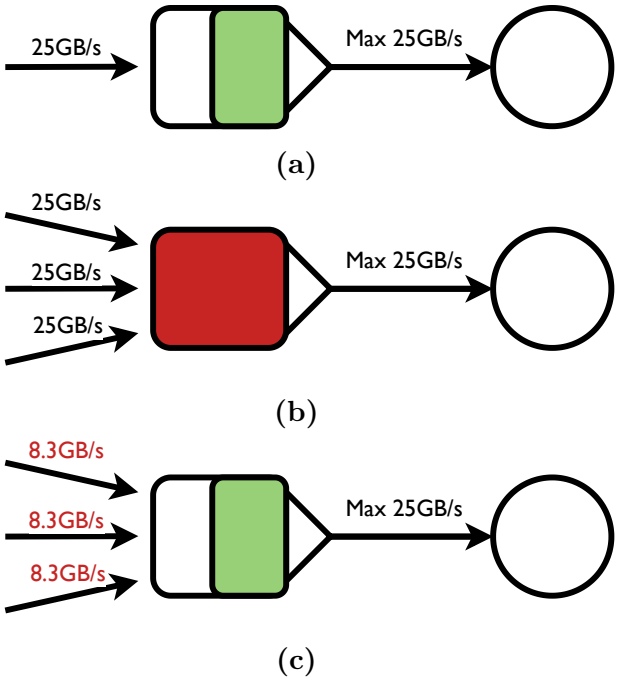


Figure 6.1: Example visualizations of switch output buffers (rounded rectangle) connected to endpoints by a single 25GB/s link. (a) When the aggregated incoming traffic comes from a single source which can only inject 25GB/s, the output buffer remains healthy. (b) When the aggregated incoming traffic exceeds the ejection bandwidth, the output buffer becomes overwhelmed. (c) If the aggregated incoming traffic is rate-limited, upper bounded by the ejection bandwidth capability, the output buffer will remain healthy.

6.2.1 The Many-to-One Problem

Consider a single endpoint targeting another specific endpoint injecting packets using all of its available bandwidth. As long as the targeted endpoint's maximum ejection bandwidth is at least as much as what is being injected and it is not targeted by any other source endpoints, its output buffer will likely not become overwhelmed (Figure 6.1a). This is because its capacity to eject packets from the network is roughly equivalent to the rate with which packets are arriving from the source endpoint.

On the other hand, if *multiple* endpoints target a single specific endpoint and inject packets using all of their available bandwidth, it's easy for the targeted endpoint's output buffer to become overwhelmed (Figure 6.1b). This is the *many-to-one* or *incast* problem.

As packets queue on the output port, waiting to be ejected from the network, more and more arrive, eventually filling up the buffer. Once packets fill up the buffer, new packets

destined for the same target will have to wait in buffers on other switches due to lack of buffer space. Congestion is now accumulating in the network.

This is the problem that congestion control looks to solve. It seeks to reduce the rate of injection of endpoints that, as a group, target individual output ports. By reducing the collective rate of injection to be equivalent to the rate of ejection of the targeted output port (Figure 6.1c), then said port will not become overwhelmed, resulting in significantly reduced network congestion.

6.2.2 Congestion Control

The process of congestion control (also referred to as congestion management) employed by the work in this chapter follow a three-phase approach. Congestion is *detected*, its *cause* is identified, and the source is *abated*. These three modules work together to identify problematic traffic patterns and address the problem at the source without adversely affecting non-aggressor endpoints and applications.

Congestion Detection Before network congestion can be addressed, it must first be detected by the system. Without an accurate and responsive mechanism to accomplish this, the congestion issue will continue to grow as the aggressor ranks continue to inject traffic without restriction.

In order to reduce the amount of time that problematic traffic can continue to propagate unabated, a congestion control system might employ a policy to detect warning signs of congestion. If a system waits until the congestion has become problematic to enact its abatement policy, then it will be that much harder to address. In the meantime, other applications in the network will be experiencing communication interference as network resource contention remains high until the abatement policy can return the network to a normal operative state.

Congestion Causation Detecting when and where congestion exists in the network is only one part of the pre-abatement control phase. The system needs to determine *what endpoints* are responsible for creating the buffer backlog. Without this information, the congestion control mechanism might target non-problematic ranks and adversely affect non-aggressor applications.

Congestion Abatement The method of action utilized in congestion control implementations for this work is through of the process of abatement. Aggressor endpoints identified by the congestion causation module receive an abatement signal, ordering them to throttle their rate of injection.

If the rate of injection of all identified aggressor ranks is low enough that they, collectively, cannot overwhelm the ejection capabilities of any one endpoint, then the negative effects of the otherwise problematic traffic pattern can be mitigated.

6.3 Design

The design of ***CODES-CC2*** can still be mapped to the three tines discussed in Section 5.1.1: Congestion Detection, Congestion Causation, and Congestion Abatement. In short, these three branches observe, investigate, and enforce congestion control policies, respectively.

Inspiration for this design was obtained from [72], where the authors note that the HPE Cray Slingshot Interconnect, a 1D Dragonfly network featuring HPE Cray's 64-port Rosetta switch, has the ability to track all packets in the network between any two pairs of endpoints. The authors also note that they leverage acknowledgement notifications working toward this goal for the purposes of congestion control.

CODES-CC2 and its implementation were designed to make the control loop of detection, causation, and the resulting abatement action as short as possible. The longer the duration of this control loop, the more challenging it can be to abate problematic traffic patterns.

6.3.1 Congestion Detection

The approach that ***CODES-CC2*** takes toward congestion detection is explicitly distributed. There is no overarching supervisory management system that makes decisions off of aggregated data. In the distributed approach, each switch is responsible for determining whether or not they are experiencing congestion.

Each switch keeps track of how many – and origins of – packets that are currently located anywhere on said switch, on each output port buffer, and each virtual channel on the switch. If this usage crosses a configurable 'threshold of congestion', then the switch considers that port to be congested. Setting this threshold low will allow the system to detect

congestion early enough to effectively enact its abatement policy. If this threshold is set too high, then congestion can be detected too late, and port buffers can become overwhelmed before abatement signals can be sent.

The switch will consider that port congested until its output buffer usage crosses a configurable 'threshold of decongestion', which is strictly lower than the one for detecting congestion. Once a port buffer becomes decongested, normal signals are sent by the switch to any ranks that it had previously sent abatement signals to.

The lower the value of this threshold, the more sensitive the congestion detection on the switch will be. The goal of this approach is to try and recognize congestion in its early stages and enact a control strategy before it has a chance to overwhelm any specific output buffer and spread to other routers.

6.3.2 Congestion Causation

As mentioned in the previous section, every switch keeps track of how many packets are located on the switch with varying degrees of specificity. A key note is that it also pays attention to where each packet originated. Once congestion is detected, the next step is to identify the terminals responsible for creating said congestion.

The version of congestion control implemented in Chapter 5 approached this feature with broad strokes. This meant potentially negatively impacting innocent ranks of a job that weren't contributing to the congestion. ***CODES-CC2*** attempts to address this issue by targeting specific ranks of a job that it knows for certain are contributing to specific localized congestion.

When a switch detects localized congestion, it will analyze how many packets and from whom are causing the observed congestion. It will then send an abatement signal to the ranks it recognizes as *aggressors*. The set of aggressor ranks is a subset of the ranks with packets located in the congestion location: whether that be the entire switch, a specific port, or a specific virtual channel. This subset, depending on the configured precision, may be equal to the original set. For example, a switch could target ranks with packets from jobs making up at least some percent threshold of measured traffic. If this percentage was, for instance, zero, then all ranks with a packet in the location of congestion will be sent an abatement notification.

The impact of incorrectly identifying a rank as an aggressor relies heavily on how the

strength of abatement is tuned.

6.3.3 Congestion Abatement

The mechanism for congestion abatement in ***CODES-CC2*** is very similar to that of ***CODES-CC***. Specifically, it works by throttling the rate of packet injection available to targeted ranks.

Previously, in ***CODES-CC***, this was tuned by a static throttling value applied to *all* ranks of an identified aggressor job. While this was capable of improving an aggressors' impact on a non-aggressor job, it is weakened by its rigidity. It might apply a throttling strength that's too heavy or too weak. It applies the same throttling strength to every rank of a job, even if that rank hadn't injected any packets during the time that congestion was recognized.

CODES-CC2 addresses this by forcing terminal ranks to self-limit their rate of injection. In order to accomplish this, terminal ranks are made aware when packets that they injected into the network are ejected from the network. Based on public patents for congestion control on Cray Aries-based networks, limiting the rate of injection to be equal to the rate of ejection is a known mechanism for congestion abatement [67]–[69]. This concept is featured in the design of ***CODES-CC2*** as well.

The reasoning behind this decision is that the rate of ejection for a specific terminal's packets is the network's capability to facilitate the communication injected by said terminal. If packets injected by a terminal do not encounter injection, then the rate of ejection should not be significantly different from the rate of injection. If, however, packets injected by a terminal do encounter congestion, packets will be slowed as they wait in queues of increasing length resulting in a slower ejection rate.

If N endpoints target a single destination, then the total ejection bandwidth will be split amongst them with an average of $1/N$. By limiting each incast participant's injection bandwidth to their known ejection bandwidth, the target will not be overwhelmed.

Each terminal is notified when a packet is ejected from the network. This allows terminals to adjust their rate of injection to be equal to their rate of ejection should they be flagged for abatement. This also allows for the strength of abatement to self-adjust as the network becomes less congested. As a result, terminals that are flagged for abatement despite playing a minimal role in any observed congestion have a pathway to self-adjust should their

rate of ejection reflect their minimal contribution to congestion.

6.4 Implementation

The objective of ***CODES-CC2*** is to be simple to configure, sensitive, and responsive. A major caveat to ***CODES-CC*** was the high number of configurable parameters, which made it difficult, if not impossible, to find settings that worked well in the general case. Thus the few parameters used to tune the system should be able to allow for some control of the sensitivity and rate of response. A more sensitive system would react to lower levels of traffic as potentially problematic and signal for abatement earlier. A responsive system would be capable of self-adjustment with new information faster.

6.4.1 General and Global Information

There are currently just a few parameters that require tuning in ***CODES-CC2***. The length of the bandwidth monitoring window, the latency of control notification messages, and the buffer capacity monitoring thresholds for determining when abatement should be enacted.

All pseudo-randomness that may be applied by ***CODES-CC2*** samples from RNG streams that are independent of any other LP behavior.

6.4.1.1 *Bandwidth Monitoring Window Length*

The degree of responsiveness of the system is determined by the granularity of the bandwidth monitoring time window. The shorter the window, the faster the system is able to adjust injection bandwidths while terminals are abated. It is also the time window over which ejection rate is measured.

If this window is too short, there is a possibility of "overfitting" due to the discrete nature of the system. As the length of this window approaches zero, the measurement of discrete signals – like the number of packets ejected during the window – becomes more volatile and subject to large swings in amplitude over time. Conversely, if the length of this time window is too long, problematic periods of time that occurred during the time window that warranted an adjustment of injection rate may be hidden by aggregation.

It would be beneficial, then, to utilize data about the average time between injected packets in a real-world system to pick a good value for this parameter to balance responsiveness

and stability.

6.4.1.2 *Control Notification Latency*

In a real-world system, control notifications must be delivered either along some management sub-network or as packets in the primary communication network. Simulation, however, allows for this decision to be abstracted. We assume that there exists some underlying ability for congestion control notification messages to be delivered so that they are not interfered with by traffic on the primary network. As such, we have fine control over the latency of congestion control notifications which allows for further investigation into how different amounts of latency affect the system's performance.

6.4.1.3 *Switch Capacity Threshold Monitoring*

Specifically, there are three thresholds for capacity that are monitored with increasing degrees of specificity:

Switch Congestion Threshold This is the percentage of all possible buffer capacity that must be utilized before abatement signals are sent. When this threshold is crossed, any terminals with packets identified as aggressors in any location on a particular switch will be flagged for abatement. For brevity, the behavior of the system utilizing this threshold monitoring degree has not yet been analyzed.

Router Port Congestion Threshold This is the percentage of any single port's buffer capacity that must be utilized before abatement signals are sent. When this threshold is crossed, any terminals with packets identified as aggressors in a particular switch port will be flagged for abatement. For the analysis provided in this work, this is the specific degree of locality monitored by the congestion control system. Additionally, the system can be configured to only monitor output ports – where packets are ejected from the network to their destination terminals – and all following analysis has this enabled.

Router Port Virtual Channel Congestion Threshold This is the percentage of any single port virtual channel's buffer capacity that must be utilized before abatement signals are sent. When this threshold is crossed, any terminals with packets identified as aggressors

in a particular virtual channel will be flagged for abatement. For brevity, the behavior of the system utilizing this threshold monitoring degree has not yet been analyzed.

6.4.2 Router Local Controller

All behavior for congestion control on the part of a switch is performed by the existing switch LP that represents it. This includes the forward and reverse handling of all congestion control-related state changes and events.

6.4.2.1 *State*

To organize and efficiently keep track of all packets with increasing levels of location specificity – switch, port, VC – a tree data structure was created. When a packet is received by a switch, the information that describes the packet – including source terminal and application IDs – is placed into this structure recursively. The root of this tree structure contains all information about all packets on the switch; it also contains pointers to child structure nodes that contain packet information for each port on the switch. Those child structure nodes have only the information about packets on the port, which they represent as well as pointers to child structure nodes which contain packet information for each virtual channel on the port. The data located at any one node is equivalent to the union of all data of its children.

This allows the capacity of varying levels of locale specificity to be monitored.

6.4.2.2 *Behavior*

The switch local controller has three major tasks: monitor the capacity of its buffers as described above, make decisions about its own state-of-congestion, and send abatement-related signals to terminal local controllers.

When a switch local controller registers that one of its locale-specific buffers reaches some configured threshold, it immediately analyzes all packets of that specific locality. For instance, if a single port's capacity reaches its congestion detection threshold, then it will query all packets on that specific port, regardless of which virtual channel on said port they are located on.

After the threshold has been crossed, the switch local controller recognizes that the buffer has fallen into a state of congestion and will begin the congestion causation procedure

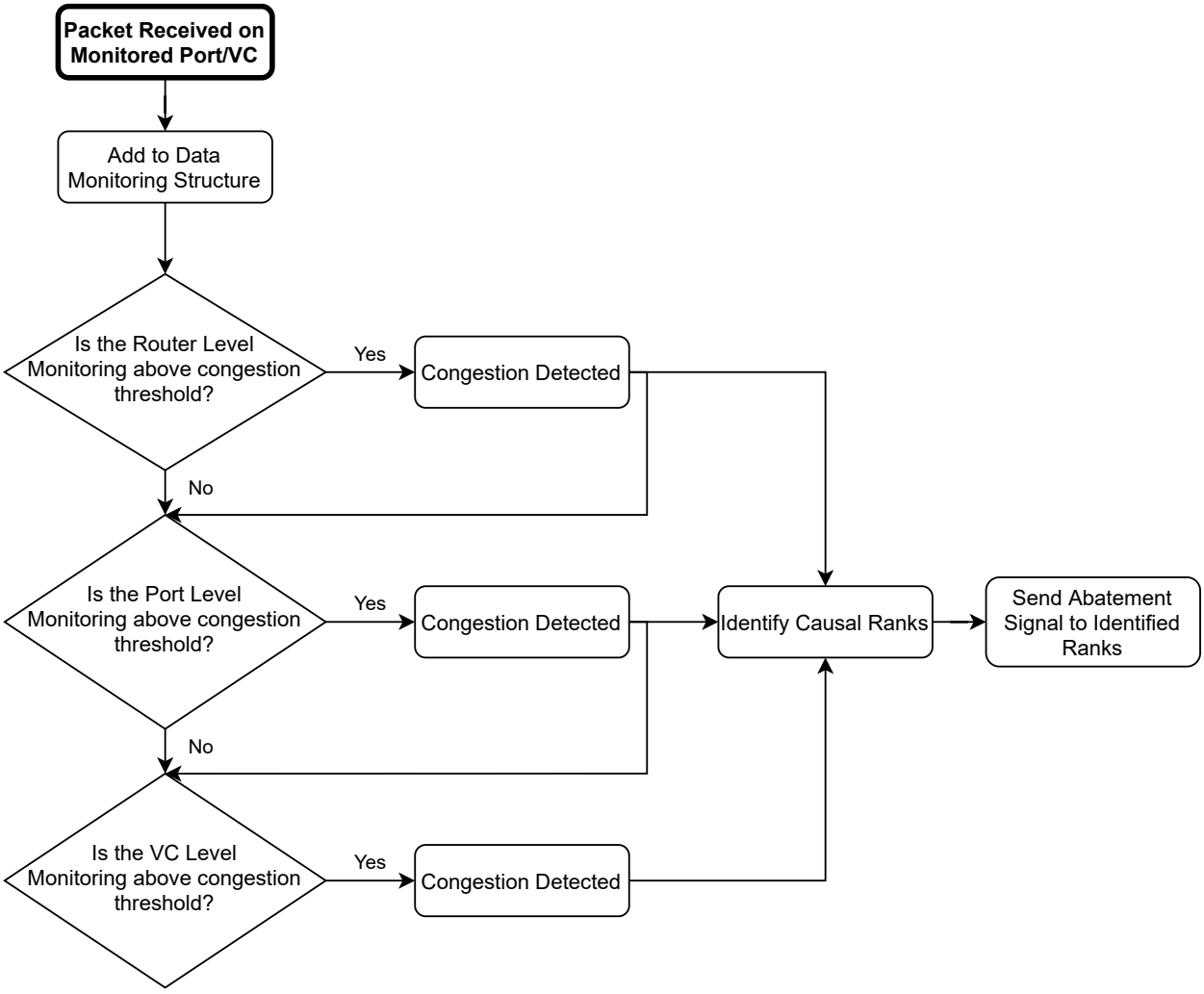


Figure 6.2: Flowchart describing behavior of switch local controller upon receipt of a packet on a monitored port with a given VC.

described in Section 6.3.2, producing a set of terminal IDs that are recognized as being aggressors.

Each terminal in the set of aggressor terminal IDs will receive an abatement signal message from this switch. Those terminals will be ordered to enact their abatement protocols if they are not already being ordered to abate by another switch local controller.

When the switch local controller recognizes that the specific locale's capacity reaches a pre-configured de-congestion threshold, it will send normal signals to all previously abated terminals from that locale. The terminals process this normal signal and will return to normal injection unless they are still abated by another switch local controller.

There is one caveat as to when abatement can end after de-congestion has been identified.

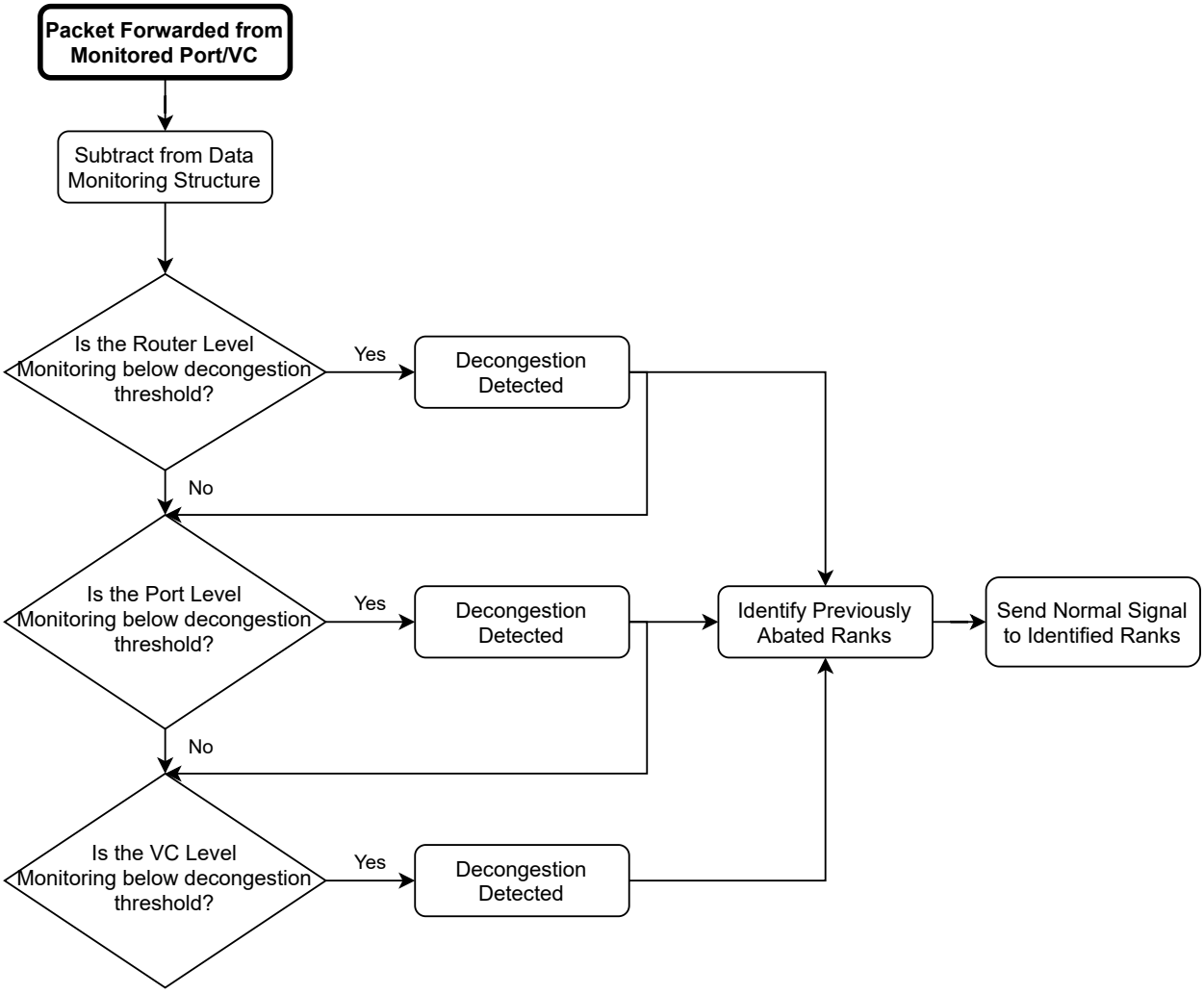


Figure 6.3: Flowchart describing behavior of switch local controller upon forwarding of a packet from a monitored port with a given VC.

Routers ordering abatement will calculate a minimum amount of time that that abatement must last before any normal signal can be sent. Upon detection, this is roughly equivalent to the amount of time it would take for the switch to reach the de-congestion threshold. However, this expiration time is extended for each packet that it receives while congested. There can be a small additional penalty multiplier that extends the duration of abatement for each byte of detected congestion allowing for the congestion control system to prevent 'bursty' aggressors from circumventing abatement protocols.

This behavior is triggered by specific occurrences in the simulation. When a packet 'receive' event occurs, the switch local controller will enact the behavior described in Figure 6.2. Conversely, when a packet 'send' event occurs, the switch local controller will enact the

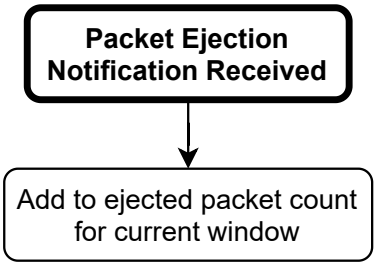


Figure 6.4: Flowchart describing behavior of terminal local controller upon receipt of an ejected packet notification.

behavior described in Figure 6.3.

6.4.3 Terminal Local Controller

All behavior for congestion control on the part of a terminal is performed by the existing terminal LP that represents it. This includes the forward and reverse handling of all congestion control-related state changes and events.

6.4.3.1 *State*

The terminal local controller has state, which allows it to know how many switch local controllers have ordered it to abate. It also keeps track of the ejection rate over a static epoch window; using that, it can then calculate the ejection rate over the course of that window.

6.4.3.2 *Behavior*

The terminal local controller has three minor tasks: send packet ejection notifications, track ejection rate of packets sourced by it, follow abatement orders from switch local controllers.

Terminal local controllers communicate with each other so that each terminal can know exactly how quickly packets sourced from themselves leave the network. This is achieved by sending a packet ejection notification to each terminal. Each terminal keeps track of how many packets sourced from itself have been ejected from the network over a static time window (behavior shown in Figure 6.4).

In order to track the ejection rate of packets sourced by itself, it simply divides the number of packets ejected during a static time window by the duration of said time window. Thus, the granularity with which it measures its ejection rate is dependent on the length of the static time window. In order to accomplish this, each terminal sends a heartbeat message

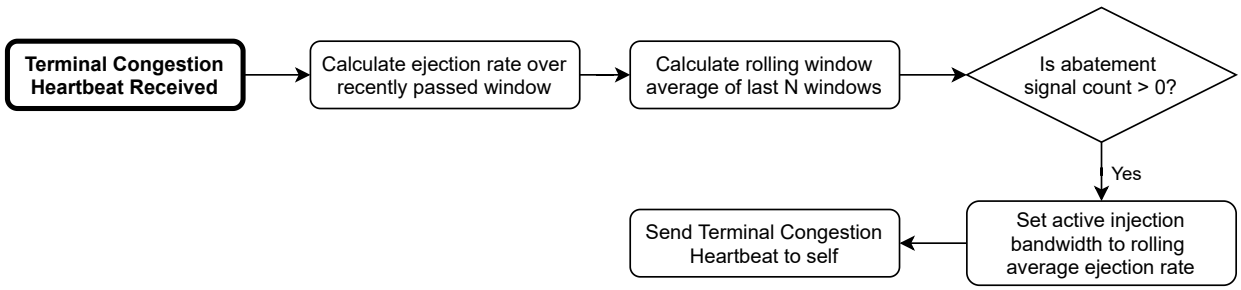


Figure 6.5: Flowchart describing behavior of terminal local controller upon receipt of a heartbeat self-message.

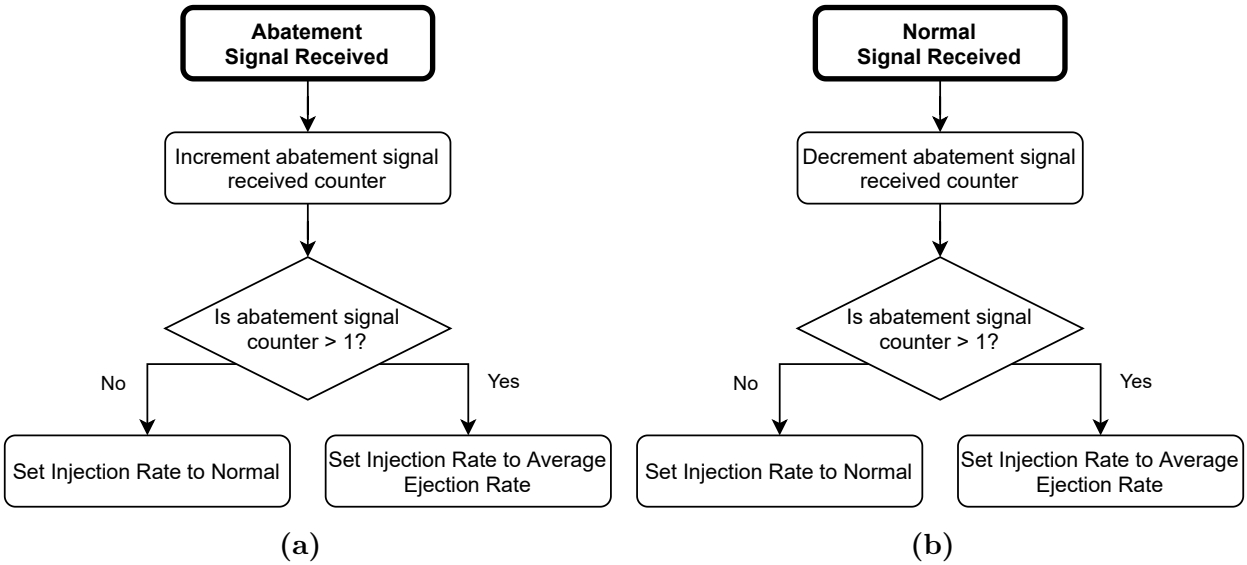


Figure 6.6: Flowcharts describing terminal local controller behavior upon receipt of abatement (a) or normal (b) signals from switch local controllers.

to itself (as shown in Figure 6.5) to calculate its latest average ejection rate for use when abatement is active.

The most critical task of the terminal local controller is recognizing whether it needs to throttle its injection rate based on abatement status. The terminal receives abatement notifications from each switch that classifies it as an aggressor. It keeps count of how many switch local controllers have outstanding abatement orders placed on it. As long as this count is greater than zero, the terminal local controller will be enacting its abatement protocol. When this count returns to zero, the terminal local controller will lift the abatement restrictions and return to normal operation. This behavior is shown in Figure 6.6.

When a terminal is following abatement orders, it limits its maximum injection bandwidth to be equal to its current measured packet ejection rate. It is possible that the ejection

rate is zero, which could cause starvation – this can be avoided by establishing some minimum assured bandwidth percentage if desired. As mentioned earlier, the rate at which its ejection bandwidth is updated is dependent on the length of the measuring time window; thus, the rate at which its injection bandwidth is updated is also dependent on this time window.

6.5 Evaluation

As was the case with ***CODES-CC***, ***CODES-CC2*** was originally designed to emulate the congestion control systems that may be found on future 1D Dragonfly networks like Slingshot [72]. It is worth evaluating the performance of the improved ***CODES-CC2*** against the workload sets that we evaluated the performance of ***CODES-CC*** against to verify that we have done at least as well as the older, more complicated version. ***CODES-CC2*** also addresses the weaknesses of the ***CODES-CC*** system against incast traffic patterns. The following experiments reflect that.

We also test the system with more intense workload sets on larger networks which is more indicative of what would be expected of a real-world high-performance system.

6.5.1 Environment

This work focuses on understanding the influence that many-to-one-traffic-based network congestion can have on other applications and how techniques for congestion control can be utilized to mitigate it.

We subject simulated 1D Dragonfly networks to different sets of communication workloads and traffic patterns in order to gain insight into the capabilities of a congestion control system. Specifically, we employ the Scalable Workload Models (SWM) [32] and SST Ember motif online MPI traffic generation suites to supply both well-behaved and aggressor workloads. We can subject the same workloads with and without congestion control and observe the benefits across a number of metrics. Examples of metrics that we can record include the total communication time of an application from its first message sent to its last message received as well as the end-to-end latency of each packet.

6.5.1.1 *Workloads and Traffic Patterns*

LAMMPS The SWM `LAMMPS` workload features a variety of types of communication. There are a number of messages sent both with and without waiting for acknowledgement of

receipt and Allreduce exchanges. The usage of iterative blocking sends is a traffic pattern that is particularly sensitive to congestion.

MILC The SWM `MILC` workload has less variety than `LAMMPS` but is still sensitive to congestion. It posts a number of non-blocking sends and receives, waits for them all to complete, and then completes two Allreduce exchanges.

Incast The SWM `INCAST` workload is a many-to-one traffic pattern. One endpoint in the workload is the designated recipient and each other endpoint sends a number of non-blocking messages to it. Its usage as an SWM is designed to be iteratively repeated; all communication from each iteration will be completed before the next is allowed to start.

Periodic Aggressor The SWM Periodic Aggressor workload is a combination of the SWM `LAMMPS` workload and the SWM `INCAST` workload. All ranks will participate in a single iteration of the `LAMMPS` workload, then a subset of them will enter a phase of a number of `INCAST` iterations. Once the `INCAST` iterations have completed, the workload will complete another iteration of `LAMMPS`. This workload is useful for analyzing the responsiveness of a proposed congestion control system, how quickly it can respond to a problematic traffic pattern as well as how quickly it can return to a normal state once the problem has ceased.

Fixed Pairs The SWM `FixedPairs` workload features blocking sends from one half of the workload's ranks to the other half – a simple bisection-bandwidth bound pattern. We can apply several iterations of this simple workload over the course of the simulation. If congestion is encountered by the resulting streams of data, it will very noticeably be impacted. This workload has been configured such that the senders in each pair will send significant amounts of sustained data but cannot exceed the ejection capabilities of the receivers. This traffic can, however, cause interference due to resource contention on intermediate switches.

6.5.1.2 *Network Configuration*

This work features simulated 1D-Dragonfly networks of two sizes. The smaller with 3,078 compute nodes is used for simple demonstrations of the concept of congestion control and features 342 36-port Infiniband-EDR (12.5GiB/s) class switches, arranged in 18-switch

groups each with nine compute node and global links. This arrangement provides nine global links between each switch group.

The larger, highly inter-connected, 8,192 compute node network is used for more realistic-scale experiments. This network features 512 Infiniband HDR (25GiB/s) class switches with 64-ports, providing sixteen 32-switch groups with 32 global connections between each switch group.

All job-to-node mappings are randomized across experiments to prevent any accidental localization bias on the final results. Within a given experiment (different routing algorithms or congestion control status), the mappings remain constant.

6.5.2 Revisiting Previous Experiments

As ***CODES-CC2*** seeks to improve upon the implementation and results of ***CODES-CC*** in Chapter 5, we apply some of the same experiments but with ***CODES-CC2*** instead.

6.5.2.1 *Trace Based Experiment*

Just as performed in Section 5.4.2.1, we utilize a combination of two DUMPI HPC application traces, AMG1728 and MG1000, as well as a synthetic uniform random aggressor workload of 350 ranks to a 1D Dragonfly network with 3,078 compute nodes. As mentioned above, this section is purely to show that ***CODES-CC2*** does not fare worse than ***CODES-CC*** in the simple case with an obvious synthetic aggressor.

We analyze the performance of the combined two trace workloads with and without ***CODES-CC2***. We do the same with the combination of the two trace workloads simultaneously with the synthetic aggressor. The simulated application performance impact of the feature is measured with the maximum communication time and packet latencies.

Figure 6.7 shows a comparison of the above-mentioned workload sets with and without congestion control enabled. We observe that without a synthetic aggressor, the two workloads are well-behaved, and congestion control has minimal impact on the communication timings of the facilitated applications. In contrast, when a synthetic aggressor job is introduced, the primary workloads, AMG1728 and MG1000, experience significant interference, which is nearly eliminated when congestion control is enabled. The ***CODES-CC2*** system recognizes the high amount of traffic being injected into the network by the synthetic ranks and limits their rate of injection to reduce its impact on the primary workloads.

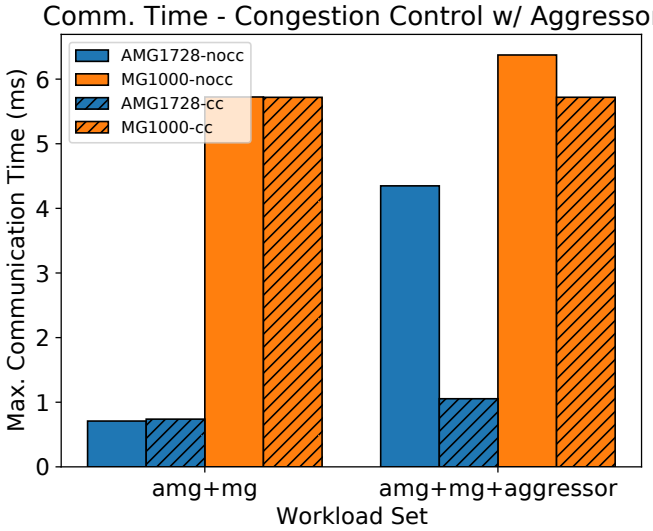


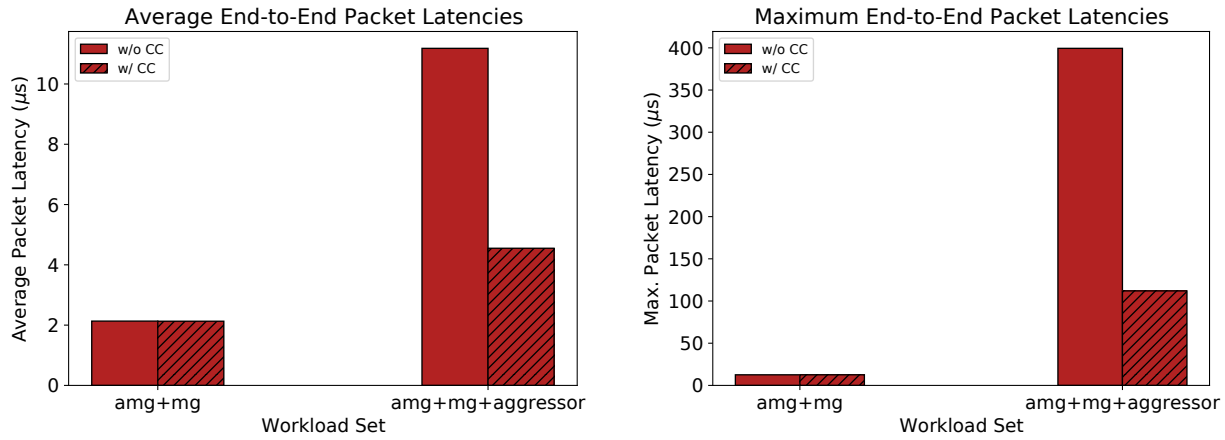
Figure 6.7: Comparing two workload sets, one without and one with a synthetic aggressor job. Shaded bars represent application performance with congestion control enabled.

Another effect of congestion control is the reduction of packet latencies in the network. Figures 6.8a and 6.8b show the significant impact that congestion control can have on reducing overall packet latencies. Specifically, it was capable of reducing the average packet latency down much closer to the baseline and reduced the maximum packet latency down by a factor of four. This reduction in packet latency is indicative of the network's ability to facilitate the same primary network traffic whilst keeping application interference lowered in the presence of other, aggressive, applications.

As this is a repeated execution of the experiment performed in Section 5.4.2.1, we can directly compare the found results. One should note that the slight variation in numerical data is a result of unrelated simulation model refinements made in the time between ***CODES-CC*** and ***CODES-CC2***. We find that the ***CODES-CC2*** system makes significant improvements over its predecessor, both in application communication time and overall packet latencies across the board.

6.5.2.2 *SWM Based Experiments*

In Section 5.4.2.2, we applied a combination of two SWM workloads and analyzed the performance of the ***CODES-CC2*** system. Specifically, the analyzed workloads included an SWM LAMMPS2048 workload and an incast traffic pattern-based workload. For this



(a) **(b)**

Figure 6.8: Comparing the mean (a) and maximum (b) end-to-end packet latencies with and without *CODES-CC2* enabled. *CODES-CC2* made minimal-to-no impact on the workload set without the synthetic aggressor but significantly reduced latencies in contrast to when an aggressor workload is introduced.

chapter, we explore different, additional workloads than that which we applied during the evaluation of ***CODES-CC***. This is because ***CODES-CC*** was observed to not handle incast aggressors without a significant alteration that made any decided abatement permanent in addition to manually adjusting the abatement strength. The experiments that followed those adjustments served as a proof-of-concept that it is *possible* to improve the performance of jobs in the presence of an incast aggressor with congestion control.

CODES-CC2 attempts to leverage the lessons learned in Chapter 5 into a simpler, more capable system. It does not require any manual interventions to strengthen or weaken the abatement protocol - nor does it permanently penalize ranks identified as aggressors.

Similar to Section 6.5.2.1, the simulations exhibited in this section are configured identically to those performed in Section 5.4.2.3 to show that ***CODES-CC2*** is at least as capable as ***CODES-CC***, if not better. Specifically, we apply a single iteration of the SWM LAMMPS2048 workload with an interfering INCAST100 workload on a 3,078 node 1D Dragonfly network. Figure 6.9 shows the result of applying congestion control to this workload set. We observe a significant reduction in communication time of the primary LAMMPS workload, lining closely with the observed performance of ***CODES-CC*** in Section 5.4.2.3.

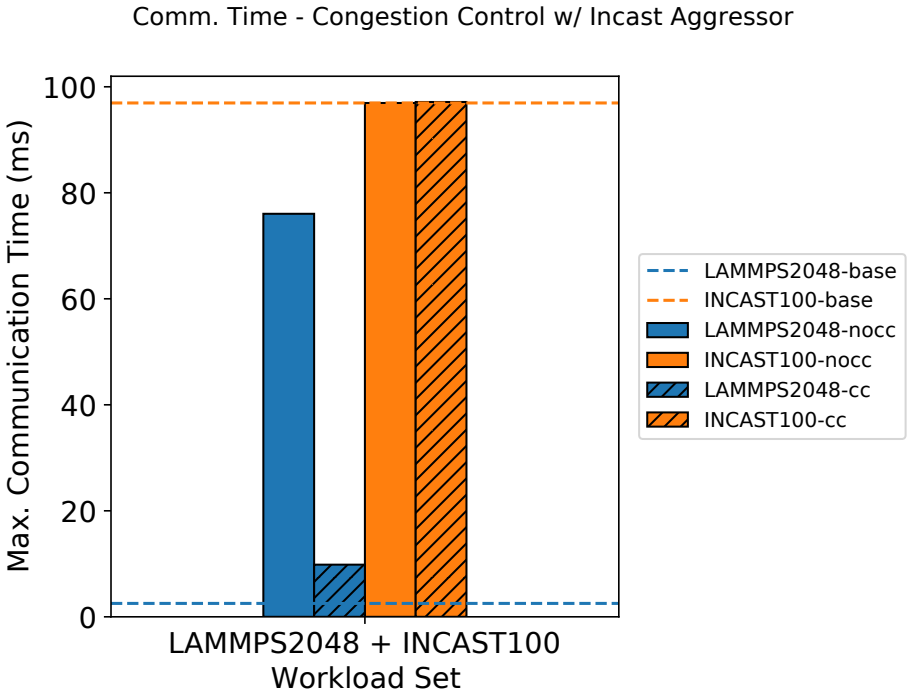


Figure 6.9: Comparison of maximum communication times of a single LAMMPS2048 and an interfering INCAST100 workload with and without congestion control enabled. Also shown is the baseline performance of each individual workload if executed on the network by itself.

6.5.3 Further Exploration

The ***CODES-CC2*** system is much more capable and flexible than ***CODES-CC***. This allows us to try workload sets of varying configurations and sizes without needing to spend a great deal of time to manually tune and adjust the congestion control system to account for it. This makes it much more likely to be reflective of a system placed on a real-world system. As such, the workloads we have applied to it are in greater numbers and on a more realistically sized 1D Dragonfly network of 8,192 compute nodes.

To evaluate the performance of a proposed congestion control mechanism, we present several experiments featuring well-behaved workloads sharing a network with varying numbers of aggressively competing many-to-one incast traffic patterns.

These experiments are meant to show the capabilities of these proposed congestion control systems but, more importantly, show how simulation can be a helpful tool in understanding the dynamics of a given technology. Simulation is not, by itself, a replacement for experiments with real-world interconnects but is instead a way to explore novel or existing

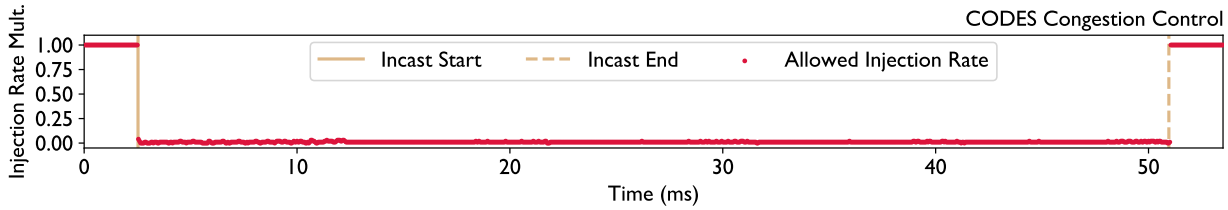


Figure 6.10: Allowed injection rate over time of rank 0 in a `PeriodicAggressor` workload which has several iterations of an aggressive 100 rank incast between two iterations of a non-aggressive 2048 rank `LAMMPS` pattern. *CODES-CC2* features are activated immediately upon detection of the problematic pattern, throttling by $\approx 1/N$, and the return to normal once it ceases.

behaviors and the dynamics of the system they operate in.

6.5.3.1 *Responsiveness*

To be able to adequately address congestion once it is determined to exist in the network, a system for congestion control must react quickly and with adequate strength to throttle the identified aggressors. The faster that the system can respond to a problematic traffic pattern, the less impact that the resulting congestion will have on nearby application traffic.

In order to demonstrate the responsiveness of the ***CODES-CC2*** module given changing application behavior, a `PeriodicAggressor` workload was run in isolation on a simulated 3,078 node dragonfly network. This workload by itself will perform an aggressive incast pattern between two well-behaved iterations of `LAMMPS` with 2,048 ranks. After the first `LAMMPS` iteration has finished, the first 100 ranks perform several iterations of a strong 99-to-1 `INCAST`. After those have all finished, the application returns to a well-behaved `LAMMPS` behavior.

Figure 6.10 shows the results of this experiment. Specifically, it depicts the allowed injection rate of rank 0, which participates in both the `LAMMPS` phases as well as one of the 99 sending ranks in the `INCAST` phase. Near instantaneously, as the `INCAST` phase begins, ***CODES-CC2*** begins throttling the participants of the many-to-one pattern down to one-hundredth of their maximum injection bandwidth. Similarly, very quickly after the `INCAST` phase ends, ***CODES-CC2*** returns the previously abated participants to their normal rate of injection.

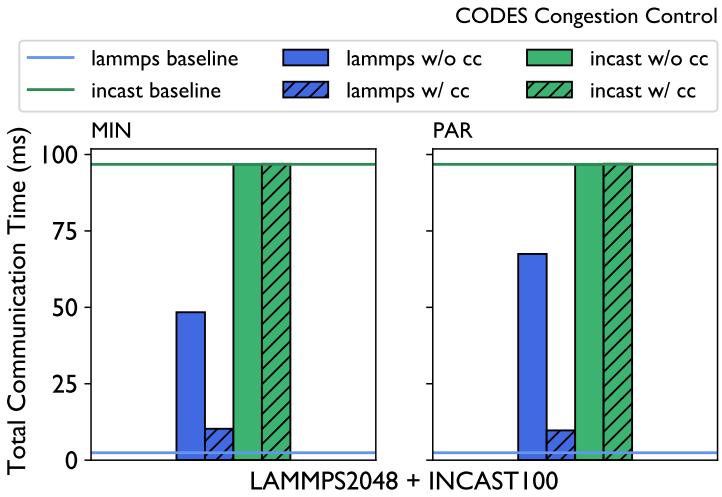


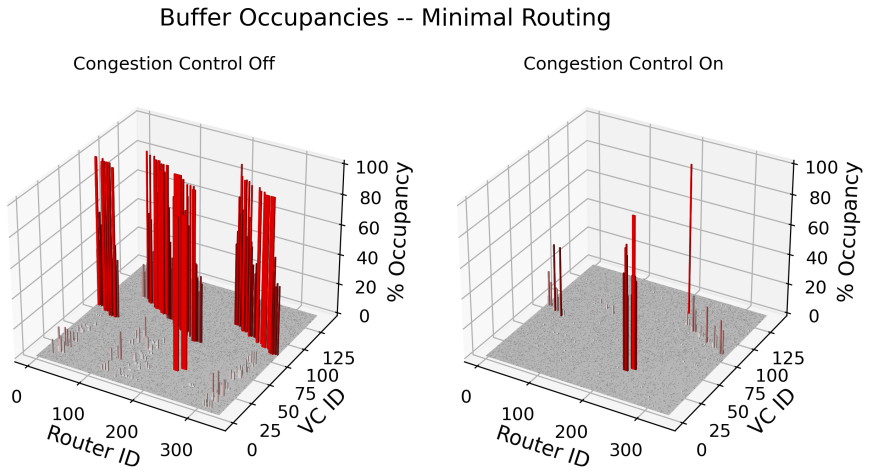
Figure 6.11: Communication time of a 2,048 rank `LAMMPS` subjected to an aggressive 99-to-1 `INCAST` on a CODES simulated 3,178 node 1D dragonfly featuring minimal and adaptive routing with and without *CODES-CC2* features.

6.5.3.2 *Effects of Adaptive Routing*

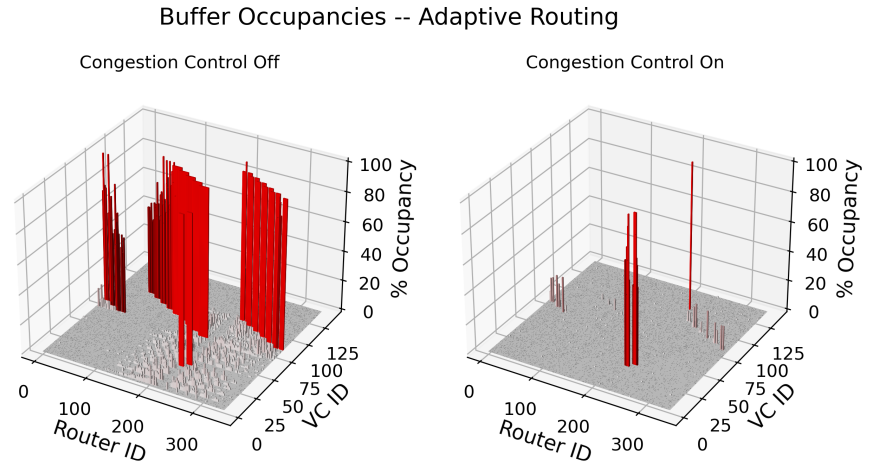
In an entirely unoccupied network, the fastest path between any two endpoints is also the path that contains the fewest number of visited switches. This path and any of the same total length are referred to as minimal routes. In an occupied network, however, the shortest path may not always be the fastest. Pockets or hot spots of localized congestion dispersed throughout the network can cause delays along given minimal routes. Consequently, it can sometimes be beneficial to take a *non-minimal* route [66] and attempt to re-route around observed congestion.

This can, however, have an unintended consequence with congestion-causing traffic patterns [73]. By nature, congestion-causing patterns such as many-to-one become problematic because they overwhelm the capacity of a single output buffer. They then, through a cascading effect, consume more and more buffer space further away from the ejection port and closer to the sources until the built-up back-pressure eventually causes the sources to self-limit. By enabling adaptive routing and allowing non-minimal paths as possible alternative routes, there are then more available lanes for congestion-causing traffic to occupy, which spawns more opportunities for application communication interference and resource contention.

This effect can be observed with a simple experiment on a 3,078 node dragonfly in Figures 6.11 and 6.12. We subject a 2,048 rank `LAMMPS` workload to an aggressive 99-



(a)



(b)

Figure 6.12: Snapshot of CODES network buffer occupancies 2ms (virtual time) into the simulation shown in Figure 6.11 with minimal (a) and adaptive (b) routing.

to-1 `INCAST` both with and without ***CODES-CC2*** and also with and without adaptive routing. With adaptive routing, the performance of the `LAMMPS` workload was indeed degraded. Figure 6.12 shows the buffer occupancy of each router in the network; adaptive routing opens up many more lanes of traffic to the aggressive incast pattern, which was previously limited to minimal routes. This has the effect of spreading traffic around the network, impacting data streams that otherwise would not have been. With congestion control enabled, it reigns in the aggressor, and adaptive routing can be effectively utilized again.

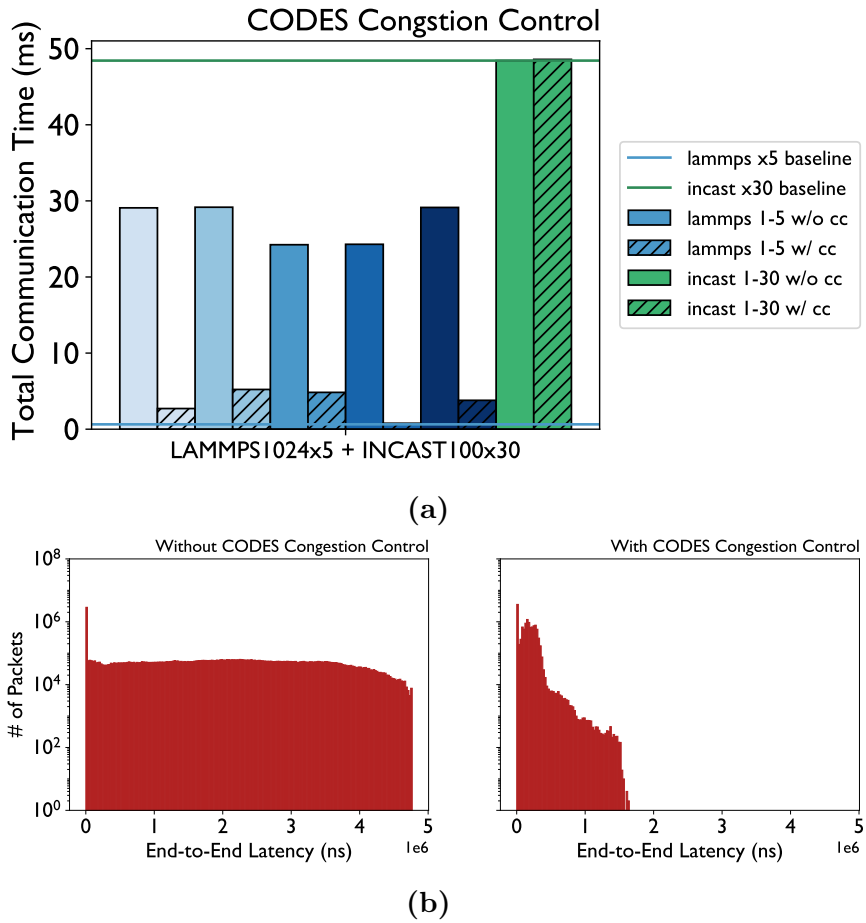


Figure 6.13: Effects of *CODES-CC2* with minimal routing on 35 simultaneously executed workloads on an 8,192 1D dragonfly network: five 1,024-rank `LAMMPS` and thirty aggressive 99-to-1 `INCAST`. (a) Per-application communication time. (b) End-to-end packet latency distributions; log-y scale.

6.5.3.3 *Throughput Restoration Study 1*

In Figures 6.13 and 6.14, we show the results of a workload set of 8,120 allocated endpoints out of an 8,192 node dragonfly network. 5,120 total ranks are comprised of five independent `LAMMPS` workloads; 3,000 total ranks are comprised of thirty aggressive 99-to-1 `INCAST` workloads.

We observe that in Figures 6.13a and 6.14a the `LAMMPS` workloads are significantly impacted by the aggressor `INCAST` workloads. Just as observed in Section 6.5.3.2, we observe that the adaptive routing fosters an even more hostile environment in the presence of the `INCAST` aggressors. With congestion control, however, the network is able to be restored to a healthier level of operation.

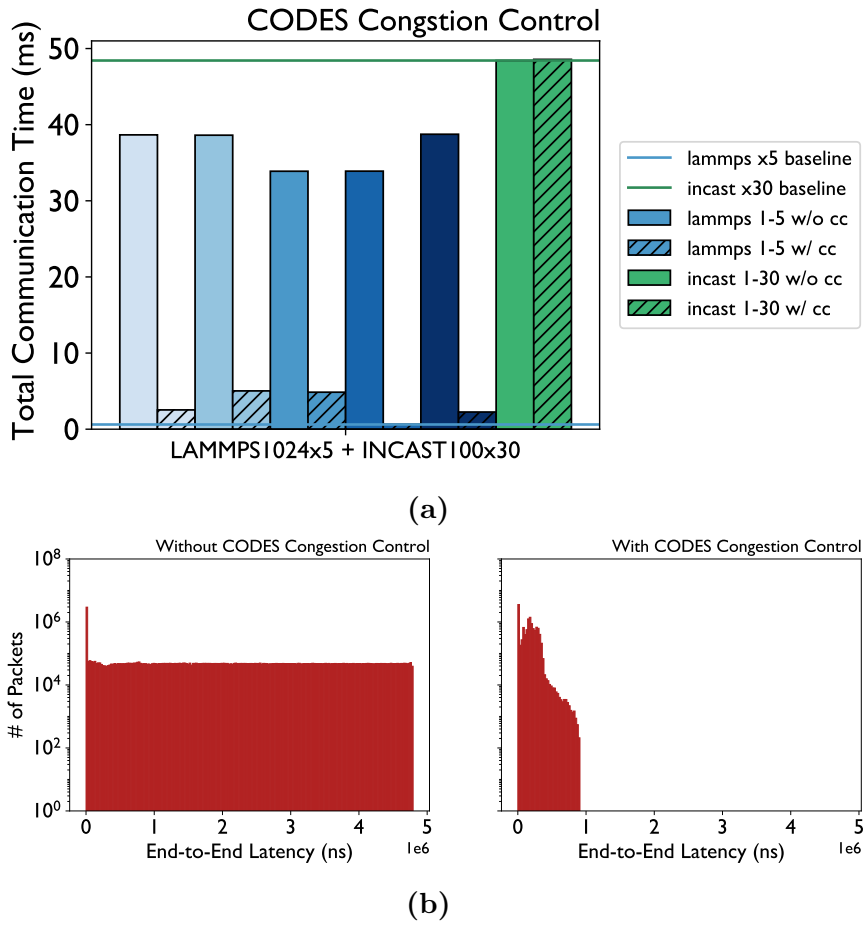


Figure 6.14: Effects of *CODES-CC2* with adaptive routing on 35 simultaneously executed workloads on an 8,192 1D dragonfly network: five 1,024-rank `LAMMPS` and thirty aggressive 99-to-1 `INCAST`. (a) Per-application communication time. (b) End-to-end packet latency distributions; log-y scale.

The same conclusions can be made from Figures 6.13b and 6.14b which shows that ***CODES-CC2*** is able to significantly reduce the length of the latency distribution tail. This corresponds to majorly reduced mean packet latencies in addition to a lower maximum.

6.5.3.4 *Throughput Restoration Study 2*

In Figures 6.15 and 6.16, we show results of a workload set of 8,192 allocated endpoints out of an 8,192 node dragonfly network. 5,120 total ranks are comprised of five independent `LAMMPS` workloads just as before; 1,792 total ranks are comprised of seven independent `FixedPairs` workloads. Two different types of `INCAST` workloads are also utilized in this workload set. Ten jobs are comprised of independent aggressive `INCAST` patterns, each with

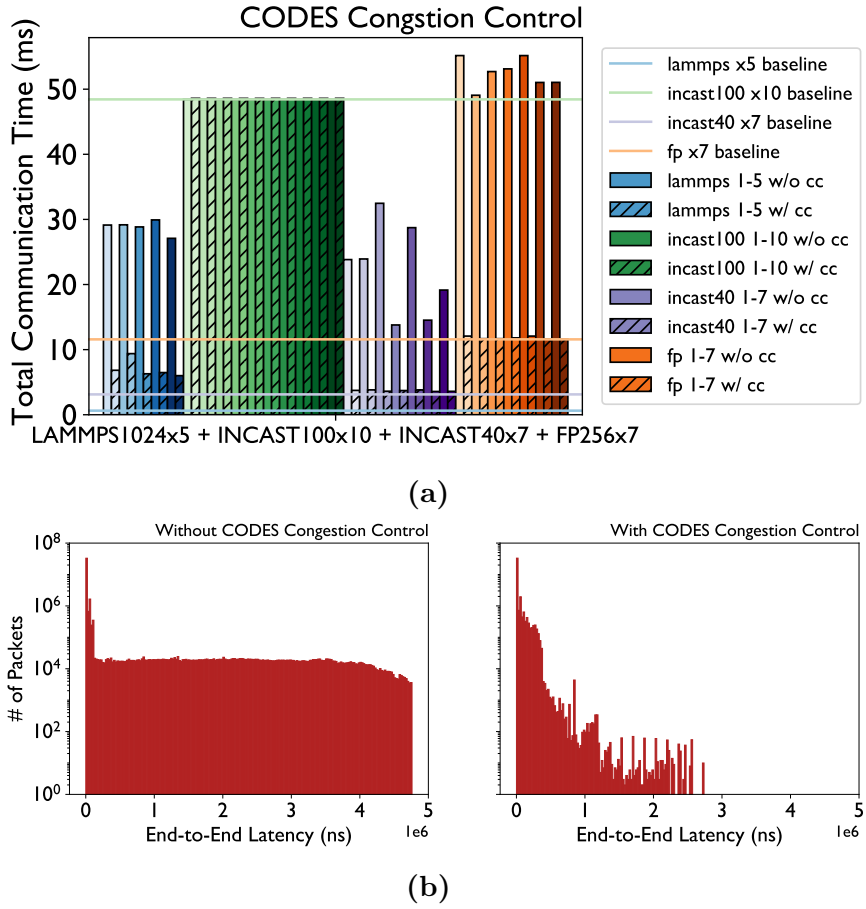


Figure 6.15: Effects of *CODES-CC2* with minimal routing on 29 simultaneously executed workloads on an 8,192 1D dragonfly network: five 1,024-rank `LAMMPS`, ten aggressive 99-to-1 `INCAST`, seven minimally aggressive 39-to-1 `INCAST`, and seven 256-rank latency-sensitive and interfering `FixedPairs`. (a) Per-application communication time. (b) End-to-end packet latency distributions; log-y scale.

100 ranks (individually identical to the one used in Section 6.5.3.2). Four jobs are comprised of smaller and less intense `INCAST` patterns of 40 ranks each.

We record that in Figures 6.15a and 6.16a, without congestion control, each job is significantly impacted by the aggressive many-to-one patterns – including the weaker `INCAST` workload. Again, we see that adaptive routing without congestion control has a negative impact on application performance. The `FixedPairs` workload, which is designed to be very sensitive and indicative of congestion, should it exist in the network, is particularly affected, unable to complete until after the aggressive `INCAST` completes. The `FixedPairs` workload, which features repeated synchronized traffic streams from specific ranks to other specific

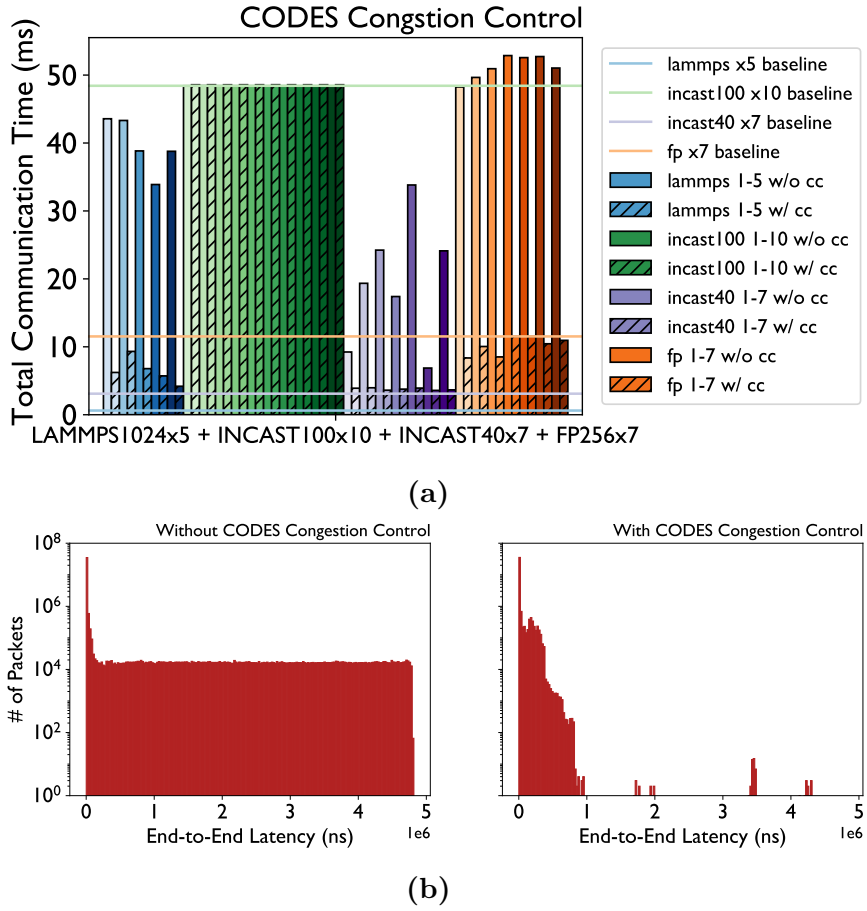


Figure 6.16: Effects of *CODES-CC2* with adaptive routing on 29 simultaneously executed workloads on an 8,192 1D dragonfly network: five 1,024-rank `LAMMPS`, ten aggressive 99-to-1 `INCAST`, seven minimally aggressive 39-to-1 `INCAST`, and seven 256-rank latency-sensitive and interfering `FixedPairs`. (a) Per-application communication time. (b) End-to-end packet latency distributions; log-y scale.

ranks, sends randomized chords of sustained traffic across the network. Because no one rank can continue to its next step until all have completed their sends, if any one stream encounters significant delays due to congestion, then the overall performance will be affected. `LAMMPS`, which has several phases of blocking collective operations, is also significantly impacted.

Conversely, with congestion control, each workload apart from the congestion-causing `INCAST` aggressors is able to complete with times much closer to their baseline "empty-network" communication times. These baseline numbers are the maximum communication time of any one workload per type operating in the network as a group: e.g. the `FixedPairs` baseline is the *maximum* communication time over all seven 256-rank `FixedPairs` workloads operating

in the network collectively without any other types of workloads sharing network resources.

The less intense `INCAST` workloads performance disparity is particularly interesting. This implies that the 40 rank `INCAST` patterns – which also have significantly reduced payload intensity – are not ejection bound but that their communication time is significantly impacted by any delay in transmission. In contrast, even without congestion control being active, the aggressive `INCAST` patterns in green are already at their baseline. Furthermore, when their injection rate is reduced to a hundredth of their maximum, they are only minorly affected. This means that the green `INCAST` patterns are significantly ejection bound, and even large delays in transmission or injection do not affect their ultimate performance.

Variations in per-job communication times within each type of application are the result of the randomly generated rank-to-node mappings.

Figures 6.15b and 6.16b shows distributions of the end-to-end packet latencies for all packets injected into the network across all applications. Without congestion control, the aggressive applications' traffic is spread throughout the network by the adaptive routing, impacting the queuing times for packets in nearly every buffer they encounter. When congestion control is enabled, the adaptive routing is able to provide short end-to-end latency times for the vast majority of packets. Outliers in the data appear to be the result of poor choices in adaptive routing as these are made based on local, not global, information on each switch.

6.5.3.5 *Throughput Restoration Study 3*

While the experiment performed in Section 6.5.3.4 showed good performance of ***CODES-CC2***, the `FixedPairs` workload, used as an indicator for congestion due to its sensitivity, can itself cause delays to other application packets because of its sustained traffic. We, thus, also simulate another workload set in a similar manner shown in Figures 6.17 and 6.18. These figures shows the effects of ***CODES-CC2*** on a workload set consisting of the same five 1,024-rank `LAMMPS` workloads, four 625-rank `MILC`, four aggressive 99-to-1 `INCAST`, and four weakly aggressive 39-to-1 `INCAST` totaling 8,180 allocated endpoints in an 8,192 node 1D Dragonfly network.

We observe in Figure 6.18a that the comparatively small `INCAST` patterns are still able to significantly reduce the overall performance of the network. This follows from the findings shown in Figure 6.11, where a single incast traffic pattern can cause significant network

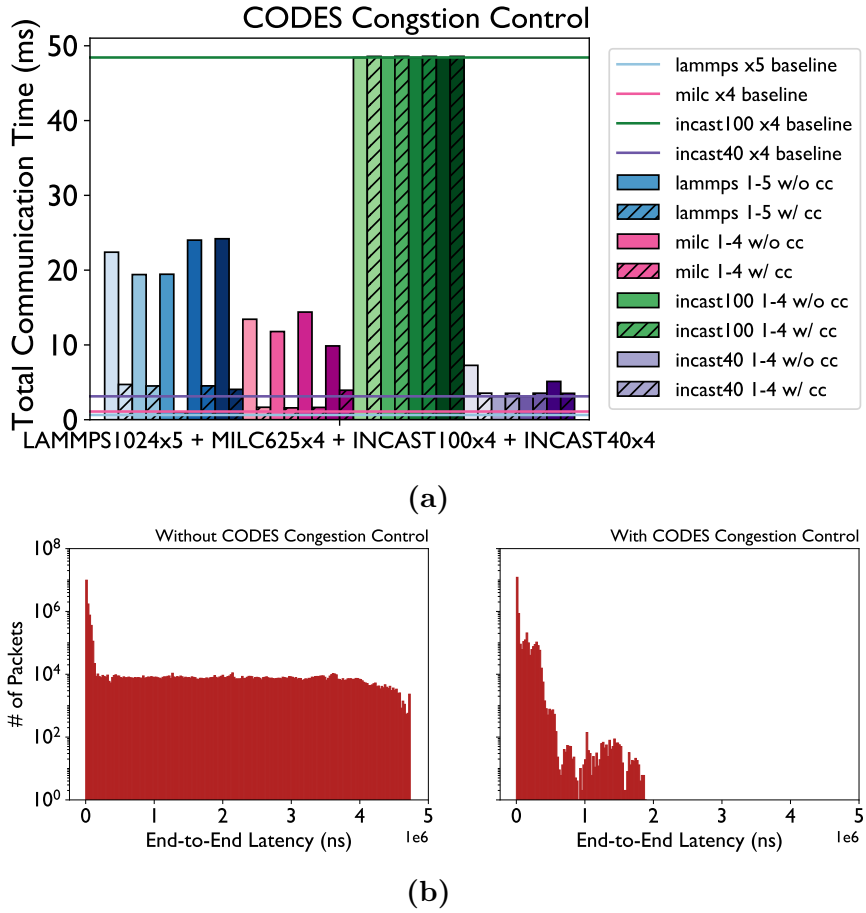


Figure 6.17: Effects of *CODES-CC2* with minimal routing on 29 simultaneously executed workloads on an 8,192 1D dragonfly network: five 1,024-rank latency-sensitive `LAMMPS`, four 625-rank latency-sensitive `MILC`, four aggressive 99-to-1 `INCAST`, and four minimally aggressive 39-to-1 `INCAST`. (a) Per-application communication time. (b) End-to-end packet latency distributions; log-y scale.

throughput problems. ***CODES-CC2***, when enabled, is able to bring all workloads much closer to their baseline by limiting the amount of traffic that these `INCAST` patterns can inject. We also note that the weaker 40-rank `INCAST` workload did not experience the same amount of interference that it had encountered in the previous study.

Figure 6.18b shows that without ***CODES-CC2***, the adaptive routing performs similar actions as before and spreads the traffic around, increasing the thickness and length of the tail end-to-end latencies distribution. With ***CODES-CC2***, more packets are able to be routed minimally and the mean and maximum packet latencies are significantly reduced.

This study demonstrates that when a network is given just well-behaved applications

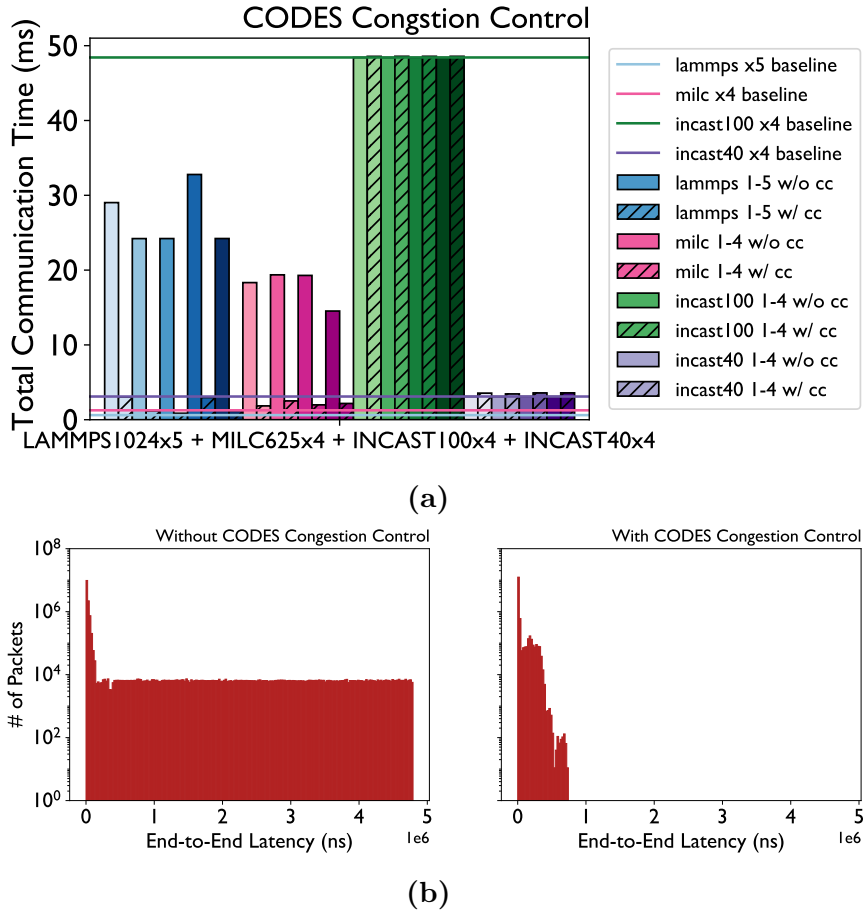


Figure 6.18: Effects of *CODES-CC2* with adaptive routing on 29 simultaneously executed workloads on an 8,192 1D dragonfly network: five 1,024-rank latency-sensitive `LAMMPS`, four 625-rank latency-sensitive `MILC`, four aggressive 99-to-1 `INCAST`, and four minimally aggressive 39-to-1 `INCAST`. (a) Per-application communication time. (b) End-to-end packet latency distributions; log-y scale.

that themselves do not create significant amounts of interference combined with the single congestion-type causing `INCAST` patterns, ***CODES-CC2*** is capable of restoring near baseline network performance.

6.5.3.6 *Exploring Effects of Congestion Notification Latency*

One limitation of the implementation of ***CODES-CC2*** is the usage of instantaneous, out-of-band communication of congestion control management messages. These include the ejection acknowledgement notifications and the abatement/normal signals from switches that instruct endpoints of congestion control orders.

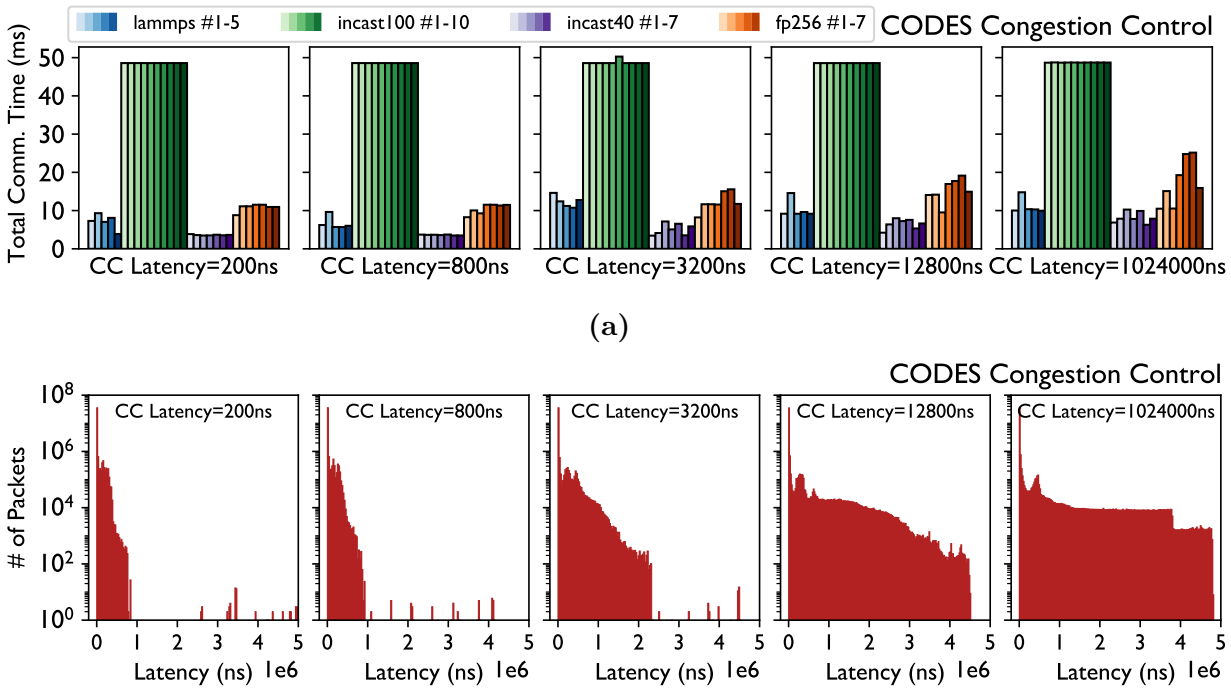


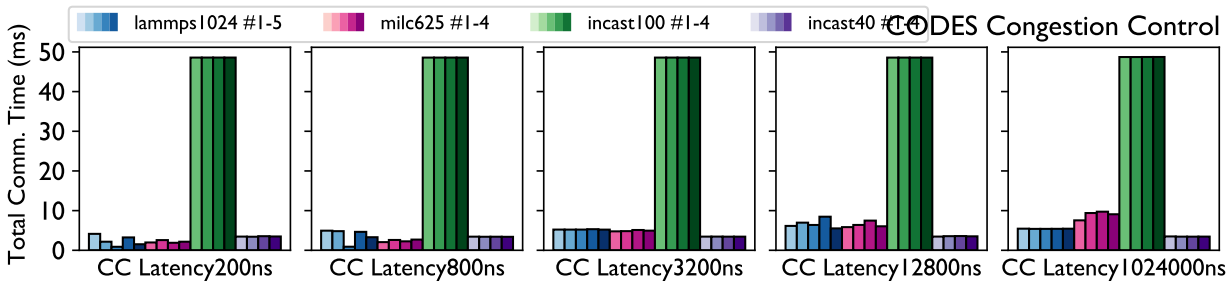
Figure 6.19: Effects of increasing the latency between the sending and receipt of congestion control notifications in *CODES-CC2* on the same experiment set used in Section 6.5.3.4. (a) Per-application communication time. (b) End-to-end packet latency distributions; log-y scale.

Instantaneous communication was first used to establish a best-case scenario for the performance of the ***CODES-CC2*** system. If it could not effectively abate the congestion by identifying and limiting the injection of problematic traffic patterns with instantaneous communication, it would be unlikely to do any better once delay is added.

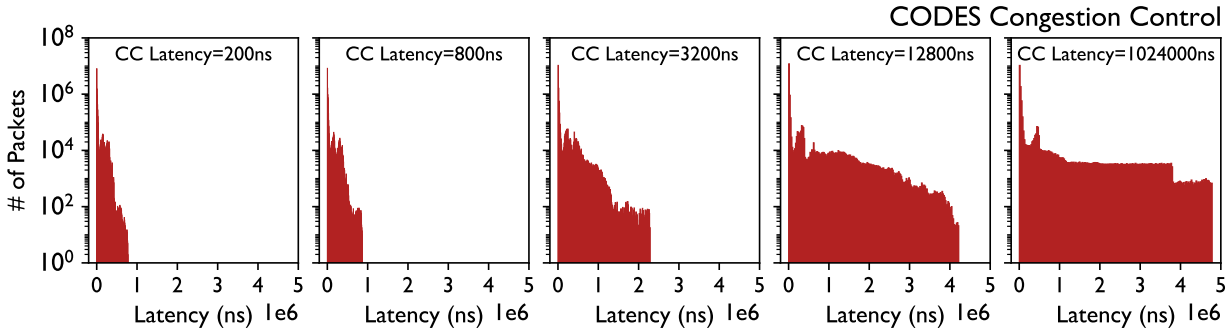
This, however, is not realistic in real-world production systems and these messages are likely to subject to their own latencies depending on how far and over what switches and buffers the messages must travel.

One advantage of simulation is the fine granularity with which developers have control. The congestion control notifications can be set to arrive at any time, including those which are physically impossible to establish best-case scenario performance results. In Figure 6.19, we do exactly that and relax the expectation of congestion control notification arrival.

Specifically, Figure 6.19 applies the same experiment exhibited in Section 6.5.3.4 but with varying degrees of congestion notification latency. We scale the latency from 200ns to



(a)



(b)

Figure 6.20: Effects of increasing the latency between the sending and receipt of congestion control notifications in *CODES-CC2* on the same experiment set used in Section 6.5.3.5. (a) Per-application communication time. (b) End-to-end packet latency distributions; log-y scale.

12,800ns and finally an extreme case of 1,024,000ns. In each of these cases, we see a weakening of the performance of the ***CODES-CC2*** features. The `FixedPairs` workload, which acts as a 'canary in the coal mine' for congestion, indicates that congestion is steadily increasing in overall severity as the congestion control notification latency increases. This is corroborated by the similarly steady increasing length in the tail of the end-to-end latency distributions. The sharp drop-off forming at the end of the tail is an artifact of the fact that congestion control is still operating in some capacity but is not capable of completely eliminating it.

However, there is still an overall reduction in the ability of the `INCAST` patterns to be able to dominate the network. Even in the worst case with significant delays in congestion control notification arrival times, the `FixedPairs` workload shows significant improvement $(2-3\times)$ over the case without congestion control shown in Figure 6.16a. Similarly, the `LAMMPS` workload observes a $3-4\times$ improvement over its no-congestion-control counterpart.

Figure 6.20 applies the same experiment exhibited in Section 6.5.3.5 but with the same

varying degrees of congestion notification latency as above. Just as before, we see the same weakening of the congestion control's efficacy as the notification latency increases.

With short congestion notification latencies, application performance was near baseline due to the lack of communication interference from the `FixedPairs` workload used in the previous experiment. In the worst-case, the `LAMMPS` workloads still have a benefit of around $3-4\times$ their no-congestion-control performance. The `MILC` workload, which before was shown to not be as sensitive to congestion as `LAMMPS`, sees only a $1-2\times$ performance boost and shows greater impact due to induced congestion.

6.6 Future Work

During the research and development of this work, there were a number of discovered possible improvements and optimizations that could be implemented into this system to refine its design or increase computing efficiency.

The congestion causation module, which identifies aggressor terminals to be abated, could employ any number of data classification procedures, including outlier detection methods such as Extreme Value Theory [74] and other data science techniques [75]. Recognizing which terminals are main contributors to congestion and which happened to have a single packet 'in the wrong buffer at the wrong time' can reduce the likelihood of false-positive aggressor classification and congestion control's impact on non-aggressor jobs.

From an implementation perspective, packet ejection notifications for every single completed packet transit generates a large number of simulation events. Because the calculation of ejection rate is performed at predictable periodic intervals, messages from one ejection terminal to an origin terminal could be summarized into a single event at the end of each measurement period.

Lastly, there is plenty of room for exploration into the effects of varying the different monitoring thresholds on each switch local controller. This work specifically focused on monitoring port capacity but could easily be expanded with the current implementation to observe how switch-level and virtual-channel-level threshold monitoring affects the performance of the congestion control system.

6.7 Conclusion

In summary, we have proposed, presented, and evaluated ***CODES-CC2***, an improved version of congestion control for use in HPC networks. The solution is distributed, delegating authority for managing congestion amongst all routers in the network as opposed to a singular supervisory controller as in ***CODES-CC***. This allows for the system to be able to quickly react to localized congestion and send abatement signals to specific identified ranks that contributed to it.

We have implemented ***CODES-CC2*** in the CODES HPC interconnect network simulation framework and evaluated many experiments showing the capabilities of the system at detecting, identifying, and addressing significant levels of congestion in real-world scale systems with a variety of workloads, both well-behaved and adversarial in nature.

We have found that the ***CODES-CC2*** system is very capable at these tasks and demonstrated the dynamics of the system with increasing levels of congestion control message latency, showing the resilience of the system and discussed its implications.

In conclusion, we believe that congestion control is a valuable asset for any HPC network with potential inter-application packet interference as it can spot the early formation of congestion and treat it at the source before it causes significant system affecting delays.

PART III

EFFECTIVE SIMULATION

CHAPTER 7
UNBIASED DETERMINISTIC TOTAL ORDERING OF PARALLEL SIMULATIONS WITH SIMULTANEOUS EVENTS

In the area of discrete event simulation (DES), event simultaneity occurs when any two events are scheduled to happen at the same point in simulated time. Simulation determinism is the expectation that the same semantically configured simulation will be guaranteed to repeatedly reproduce identical results. Since events in DES are the sole mechanism for state change, ensuring consistent real-time event processing order is crucial to maintaining determinism. This is synonymous with finding a consistent total ordering of events.

In this work, we extend the concept of virtual time to utilize an arbitrary-length series of tie-breaking values for preserving determinism in parallel, optimistically executed simulations without imposing additional bias influencing the ordering of otherwise incomparable events. Furthermore, by changing the core pseudo-random number generator seed at initialization, different orderings of events incomparable by standard virtual time can be observed, allowing for fair probing of other potential simulation outcomes. We implement and evaluate this extended definition of virtual time in the Rensselaer Optimistic Simulation System (ROSS) with three simulation models and discuss the importance of deterministic event ordering given the existence of event ties.

7.1 Introduction

Discrete event simulation (DES) is an effective method for simulating a wide variety of phenomena. It establishes any occurrence in the simulation as being an event that occurs on a simulation entity. This event alters state and occurs at a temporal coordinate in the virtual simulation time. Because simulation state only changes when an event happens, the outcome of a simulation is entirely dependent on the initial state and the set of ordered events that alters it.

It is expected that two identically configured simulations will produce identical results.

Portions of this chapter previously appeared as: N. McGlohon and C. D. Carothers, "Unbiased deterministic total ordering of parallel simulations with simultaneous events,", May 2021, arXiv:2105.00069v1 [cs.DC].

Portions of this chapter are to appear in: N. McGlohon and C. D. Carothers, "Toward unbiased deterministic total ordering of parallel simulations with simultaneous events," in *Proc. 2021 Winter Simul. Conf.*

Simulations with this property are referred to as deterministic. Determinism is not uncommon for models developed in DES. The design of a general DES engine, thus, may be of broader utilization if it can provide deterministic execution. Maintaining determinism in a simulation can help confirm that there is not additional errant, undefined behavior occurring. The defined rules of a deterministic simulation, given a singular input, result in a singular answer.

From a high-level view, maintaining determinism appears to be a simple task: process events in the order in which they are scheduled to occur. However, there are complications that can add complexity to this objective. In a parallel processing environment, a simulation is partitioned across multiple processors, with events occurring between entities that may not exist on the same processor. Because of this, the nature of inter-processor clock drift and individual processing delays, ensuring a deterministic ordering of event execution requires additional synchronization overhead.

Fortunately, this is generally a solved problem in Parallel Discrete Event Simulation (PDES) with various synchronization methods for conservative parallel and optimistic parallel execution. One complication, however, is event simultaneity: events that occur at the same coordinate in virtual time. Given two simultaneous events that each result in different, non-commutative, state changes on the occurring entity, final simulation results will naturally depend on the ordering of these two events. To make matters worse, the emergent existence of simultaneous events may not be obvious to a model developer – nor the potential consequences such as loss of determinism.

In a parallel environment, an inter-processor event may arrive at the receiving processor after an earlier timestamped event had been processed, violating the virtual time-guaranteed happens-before relation of the two events. Correction methods like reverse-computation or checkpoints can be used to revert the simulation to the last known safe state and ensure that a consistent and valid causal ordering is maintained despite this phenomenon. However, there could be a problem when two events are to occur at the same coordinates in virtual space-time. Given the encoded standard virtual timestamps alone, there is not a clear way to define which event happens-before the other, which results in a loss of determinism.

In order to regain assured determinism, a set of rules can be defined, dictating the expected ordering of two simultaneous events. Given a strictly increasing event counter on each processor that increments with every new event, then in the case of simultaneous events, we can give priority based on the sending processor ID. If that, too, is identical, then we

can give priority based on the event ID encoded from the event counter. Determinism was regained but with one major caveat: the resulting ordering is influenced by the bias of the chosen of ruleset.

Because of this bias imparted by the ruleset, it is difficult to know for certain if the resulting simulation behavior is exemplary of the simulated model's behavior as a whole or a special case determined by said ruleset's influence on event ordering. Additionally, without a deterministic total ordering of events in a parallel simulation, it is much more difficult to show that the parallel simulation is semantically the same as if it were sequentially executed. In this work we extend the definition of virtual time to include a set of tie-breaking values and devise a mechanism for unbiased arbitration of simultaneous events in a parallel discrete event simulator with optimistic execution. This is accomplished by utilizing a rollback-safe tie-breaking uniform random number generator. We develop this mechanism into the Rensselaer Optimistic Simulation System (ROSS) and exhibit its effects on simulation performance and model behavior. Because ROSS, even in optimistic execution, utilizes fully deterministic pseudo-random number generators (PRNGs), different RNG initialization seed values will yield different but deterministic event orderings, even in the presence of simultaneous events, allowing for deeper statistical analysis of simulated models.

We propose and argue three new mechanisms for finding a deterministic total ordering of events in a parallel discrete event simulation with simultaneous events:

1. Unbiased random total ordering of otherwise incomparable simultaneous events.

2. Biased random total ordering of otherwise incomparable simultaneous events, including events created with zero-delay/offset.

3. Unbiased random total ordering of otherwise incomparable simultaneous events, including events created with zero-delay/offset.

7.2 Problem Definition

For this work, we specifically utilize the Rensselaer Optimistic Simulation System (ROSS) PDES engine [24], [76], which implements the Time Warp protocol [77]–[79] in conjunction with virtual time [28]. In a ROSS simulation, entities or agents are represented as Logical Processes (LPs). These LPs are mapped to the various Processing Elements (PEs)

that may exist. Typically, there is one PE per physical MPI process that is participating in the actual execution of the simulation.

ROSS is capable of being run in three main modes of synchronization. When the simulation is to be executed on a single process, it is executed in *sequential* mode. Maintaining determinism in this mode is trivial. The challenge of determinism comes when additional PEs are added to the simulation. With additional PEs comes the advent of *remote* events, those whose destination is on a different PE than its source.

All propositions in this work are based on the following assumptions and definitions:

Assumption 1. *In a parallel environment, the arrival order of inter-processor messages is not guaranteed to be consistent across executions of a simulation.*

This is due to natural differences in local clock speed. Even the smallest perturbation can lead to enough clock drift to change the order in which any two events from different processors may arrive at a third processor. If this is not accounted for, it can lead to drastically different final results. Fortunately, this is addressed by Time Warp's virtual time approach to parallel event processing.

Assumption 2. *Every event in the simulation is encoded with a standard virtual timestamp specifying when in simulation time the event occurs.*

Without virtual time, a receiving process will have no ability to determine when in the future the event should occur. Given this encoded virtual time, it is possible for the receiving process to deduce a partial ordering of this received event and others that the process is aware of.

Assumption 3. *Each processing element in a PDES system has a priority queue maintaining a proper order of arrived events to be processed.*

Because different LPs across PEs may be scheduling events at various times in the future, we cannot assume that events will arrive in the exact order that they are to be executed – and by Assumption 1, can not be guaranteed even if one tried to make it so. Thus, each PE should manage a queue, maintaining a proper order in which events should be processed. This is almost certain to exist in any parallel simulator that implements virtual time.

In an optimistically executed simulation, events that arrive at an LP after its current definition of 'now' are referred to as *straggler* events [25]. To address this, optimistic simulations can be rolled back. After rolling back the simulation to a known safe state, the undone events can be correctly ordered in the priority queue and re-processed. None of this can be accomplished without some ordered queue.

Assumption 4. *No LP can create events at a virtual time in the past relative to its own definition of virtual time.*

This is a safe assumption as the entirety of discrete event simulation relies on the basis of maintaining causal-order; causal ordering can be formally stated as:

Definition 1 (from [28])**.** *Event A causes B ($A \rightarrow B$) if there exists any sequence of events $A = E_0, E_1, \ldots, E_n = B$ such that for each pair E_i and E_{i+1} of adjacent events either (a) E_i and E_{i+1} are both in the same process and the virtual time of $E_i < E_{i+1}$ or (b) event E_i sends a message to be received at event E_{i+1}.*

Because, in virtual time simulations, the virtual space-time coordinates of an event define the happens-before and causal relations of events, the only instance where two events are considered *incomparable* is when they have identical virtual timestamps, i.e. simultaneous events or an *event tie*.

Definition 2. *Two events are considered comparable if a consistent happens-before relation can be inferred between the two. If no happens-before relation can be inferred, then they are considered incomparable.*

Definition 3. *An event created with zero virtual-time delay between its encoded timestamp and the creating LP's definition of 'now' (timestamp of the causal event) is referred to as a zero-offset event.*

Assumption 5. *Every LP has at least one, rollback-safe, pseudo-random number generator stream solely dedicated to generating uniform random values to encode into events that it creates.*

To achieve the goal of creating an unbiased ordering of simultaneous events via a pseudo-random number generator, it is imperative that this stream be completely independent of other streams accessed in the model and deterministically rolled back. For reasons which will be

explained in detail in Section 7.3.2, it is also critical that this PRNG can be deterministically rolled back to a previous state should the events that it generated values for be canceled or rolled back.

With the above assumptions and definitions, we posit the following problem:

Problem Statement 1. *Given an optimistically or conservatively executed parallel discrete event simulation with the above assumptions, consisting of a partially ordered set of virtual-time encoded events E which may or may not contain event ties, find a mechanism that ensures unbiased total ordering of E such that any subsequent executions of the simulation are deterministic.*

7.3 Event Simultaneity

Another way to phrase Problem 1 is: given a simulation of events, find a mechanism to randomly establish the partial ordering found by a parallel simulation in a way that is deterministic and identical to the total ordered version found by a sequential execution of the same simulation. If all events in a partially ordered simulation are comparable to each other, then the partially ordered events are also considered totally ordered.

As briefly mentioned in Section 7.1, deterministic ordering of simultaneous events can be achieved by establishing a ruleset that is enforced by each PE dictating which event happens-before another. If this ruleset is precise enough to render every possible event in the simulation comparable to every other event, then a deterministic total ordering can be established from its partial counterpart. In Figure 7.1, events A, B, C and D all occur simultaneously in the simulation. How this ruleset is defined will specify how these events are ordered to occur in the simulation. The orderings of C and D are specifically of consequence due to them each operating on the same LP.

7.3.1 Ordering Bias

This ruleset would, in effect, turn incomparable – simultaneous – events into comparable events that occur *simultaneously yet infinitesimally before or after one another.* An example ruleset (Ruleset 1) would be as follows, in decreasing order of importance, cascading down to break any subsequent value ties.

This, however, will result in a single final result with likely few options that would allow users to explore alternate orderings. Perhaps with greater consequence is the amount of bias

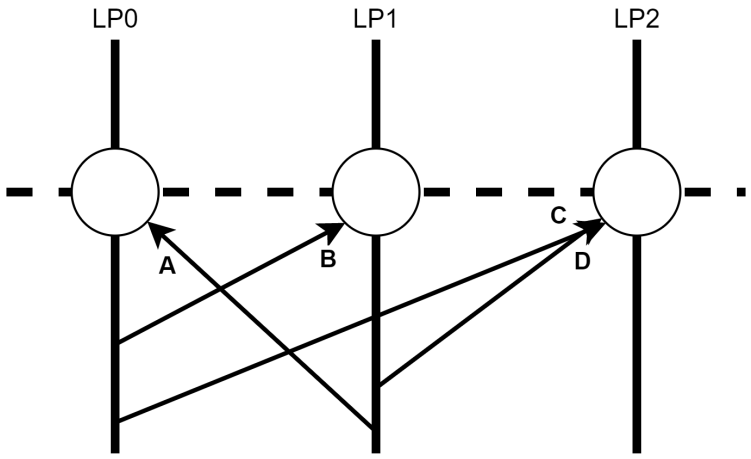


Figure 7.1: In this LP-Event diagram, all events: $A, B, C,$ and D occur at each of their respective LPs simultaneously. Determining how these events, C and D in particular, should be ordered in a final total ordering may have an effect on final simulation results.

that this ruleset instills into the simulation. In the case of a timestamp tie, it will always give priority to whatever event came from a lower-numbered processor. This means that LPs mapped to lower-numbered processors, for instance, will always take precedence should they tie and that the same simulation but with different numbers of PEs or a different mapping will likely generate different final results.

Ruleset 1 Ordering with Explicit Bias

Give priority to:

1. Events with lower virtual timestamp
2. Events with lower PE ID
3. Events with lower LP ID
4. Events with lower (per PE) event ID

Given the possibility of a two-way event tie, it is possible to relinquish the simulated model from the influences of our bias by instead relying on a fair coin flip.

With this ruleset (Rulset 2), given two tied events, A and B, there is a 50% chance of being ordered as [A,B] and a 50% chance of being ordered as [B,A]. If we can randomly generate the values necessary for this coin flip to occur, then we can, without bias, order all events in a simulation with at most 2-way event ties.

Ruleset 2 Ordering without Bias given 2-way event ties

Give priority to:

1. Events with lower virtual timestamp
2. Events that win a coin flip

This can easily be extended to n-way ties. Instead of a coin-flip, encode an i.i.d. uniform random (UR) value into an event when it is created and implement Ruleset 3.

Ruleset 3 Ordering without Bias given n-way event ties

Give priority to:

1. Events with lower virtual timestamp
2. Events with lower tie-breaking value

Given an n-way tie, each event having its own independently generated uniform random value, any specific ordering of these events constitutes a single possibility (occurring with probability $1/n!$) out of all possible permutations of these tied events. Furthermore, the probability of any two tied events being ordered a specific way with respect to each other is 50%. This method is equivalent to what is described and proven in [65] as `PERMUTE-BY-SORTING` yielding the following lemma:

Lemma 1 (From [65])**.** *Given a list of length N with N distinct i.i.d. uniform random values, a uniform random permutation can be found by assigning one value to each item in the list and sorting according to these values.*

If the probability of tie-breaker collision is negligible, then it is possible to generate a uniform, or unbiased, random permutation by sorting based on the uniform randomly generated key. It follows that by using Ruleset 3, we can extract an unbiased random ordering of an n-way event tie.

One may question whether any random ordering of these events is safe. The most important thing in ordering simultaneous events is to uphold event causality: no event caused by another can happen before the other event.

Lemma 2. *If there are no zero-offset events in a simulation, then no events participating in a tie caused any other events in said tie.*

Proof. Given the contrapositive: if events participating in the tie did cause other events in the tie, then there are zero-offset events. Suppose an event participating in the tie caused another event in the tie. Because both events are participating in the tie, the virtual time difference between the two would be zero. By definition, the caused event was scheduled with zero-offset. Thus, by contrapositive proof, if zero-offset events are not allowed, then no events in an event tie caused any other events in the same tie. □

A consequence of this lemma is that if there are no zero-offset events in a simulation, then because there is no causal linking between any events participating in the tie, no two events in the tie are ordinarily comparable. If they were comparable by their virtual timestamps, then they would not be in this tie in the first place.

Because events that are not causally related to each other and incomparable could be processed in any order without violating causality, then a random permutation of these events constitutes a valid ordering.

Proposition 1. *Providing, in addition to the standard virtual timestamp, a secondary uniform random value and comparing tied events based on that will yield an unbiased ordering of events without violating causality given no zero-offset events.*

Proof. Lemma 2 assures that no events in any event tie will have a causal relationship with any others in the same tie. Because these events, given their standard virtual timestamp, are thus incomparable, any possible permutation of these events is causally safe, and by Lemma 1, an implementation of Ruleset 3 will allow these events to be comparable and yield an unbiased ordering of events. □

For clarification, we define the combination of a virtual timestamp and any secondary values determining event priority as a *virtual time signature*. Events created in accordance with Proposition 1 that are incomparable by their virtual timestamps are comparable by their virtual time signature.

It is important to note, however, that Lemma 1 relies on *distinct* values from its uniform random number generator. By the nature of PRNGs, this is not possible. For the purposes

of our work, though, we utilize sufficiently precise PRNGs with very large periods and thus we assume the probability of generating non-distinct values is negligible.

Should this probability still be considered too high, additional tie-breaking values can be generated and encoded via supplementary, independent, PRNG streams and applying `PERMUTE-BY-SORTING` recursively to sub-lists containing PRNG value collisions. This effectively increases the bit precision of the primary tie-breaking value, making overall collision even less likely while maintaining the expected lack of bias.

7.3.2 Determinism with Randomness

We have now established that given a good choice of ruleset, we can create *an* unbiased ordering of events in a simulation. But this alone does not grant determinism. The ROSS simulator guarantees Assumption 5, allocating an independent PRNG stream on every LP with the sole duty of generating the tie-breaker values encoded into each event that that LP creates.

One valuable property of PRNGs is that given a specific seed, the sequence of values generated from it is deterministic. In a sequentially or conservative parallel executed simulation, this allows for pseudo-randomness without sacrificing determinism. In neither of these execution modes is there ever a rollback to a previous simulation state.

When executing a parallel simulation optimistically, however, the simulation must be tolerant of rollbacks. If the state of a PRNG stream, indicating the position of the 'next' value in the sequence, is not also rolled back when necessary, then a different set of numbers will be generated for events when the simulation is resumed. As a result, LP state changes and decisions based on that PRNG stream will differ from an execution where said rollbacks did not occur – a loss of determinism.

Corollary 1. *Using a rollback-safe tie-breaking uniform random value in execution of Proposition 1 will yield a deterministic unbiased ordering of events without violating causality given no zero-offset events.*

Any extra identifier that enables the comparison of two events that would otherwise be incomparable must be implemented into all parts of a system that compares two events. If, for instance, a PDES system is configured to process events in a specific order, but care is not taken to recognize any straggler message – based on virtual time signatures – and

correct it, then there will no longer be any guarantee of determinism. Any events irrevocably committed to the simulation history must be in agreement with the new, extended definition of virtual time.

7.4 Zero-Offset Event Simultaneity

Proposition 1 makes certain assumptions about the nature of the problem and the simulation environment it operates in. Specifically, it does not allow for zero-offset events to exist in the simulation.

To allow for zero-offset events in a simulation, there are some finer details that must be established to ensure that a deterministic, valid, unbiased total ordering of events can be found. Part of what makes this difficult is that the act of creating zero-offset events, by definition, creates event ties. As a result, an event that generates a new event with a zero-offset is creating a new event that, in virtual time, occurs at the same time as the event that created it. In this instance, the order in which these two events are committed to simulation history is very important.

It's also important to note that while Assumption 4 is standard for virtual timestamps, the same must also be held true for any happens-before relations inferred from an extended definition of virtual time. Furthermore, the virtual time signature of any event must be strictly greater than the time signature of its causal parent based on Lamport's Clock Conditions [80].

7.4.1 Challenge of Zero-Offset Events

We have shown with Proposition 1 that an unbiased total ordering of simultaneous events can be found with a uniform random value acting as a tie-breaker. This works by establishing an unbiased hard rule of how any two events should be ordered in the simulation. Proposition 1 worked because any arbitrary ordering of otherwise incomparable events will result in a valid simulation.

This becomes tricky once zero-offset events are introduced. Because a parent event and its zero-offset child have the same virtual timestamps, but one event *definitely* caused the other, the ordering in which these two events should be ordered in the final simulation history cannot be arbitrary.

In a solution using Proposition 1, to create an unbiased random ordering of events, an absolute comparison of i.i.d. random tie-breaking values is utilized. Given an event E that

generates a child event E' with zero-offset, it is entirely possible that the uniform random tie-breaking value of E' could be less than that of event E. By our extended definition of virtual time, this would imply that event E' *happened before* event E which is impossible since event E directly caused E'. This contradiction caused by a zero-offset event is exactly the same as what would happen if a time-traveling negative-offset event were created. This establishes the following:

Lemma 3. *No event can be created in a way that would classify it as happening before any events that caused or happened-before it.*

Without the tie-breaker value enforcing the final ordering of these events, this will execute and complete fine - but at the risk of non-determinism if there are other event-ties. But with the tie-breaking mechanism, the PDES engine will enter an endless loop of rollbacks and replays as it tries, fruitlessly, to reconcile this situation and adhere to the deterministic tie-breaking order.

If all we consider was their standard virtual timestamps, the two events are incomparable. However, they *are* comparable by the ordered chain of causality. Definition 1 does not account for zero-offset events; if we are to have a stable simulation with zero-offset events, we need to extend our definition of virtual time to ensure that the virtual time signature of $E < E'$ and so on for any descendants of event E'.

Definition 4. *An event that shares a standard virtual timestamp with another but has a lower tie-breaking value is defined to happen infinitesimally before the other. The converse implies that the other event happens infinitesimally after the first.*

Definition 5. *A zero-offset event, sharing the virtual timestamp with that of its parent, is considered to happen infinitesimally after its parent. The converse implies that the parent happens infinitesimally before its child(ren).*

To demonstrate the challenge of utilizing the tie-breaking mechanism we have established in Proposition 1, let us assume that we have some ability to ensure that Lemma 3 is maintained while using the same tie-breaking mechanism as before. Then we have nothing to prevent the possibility of generating events with virtual time signatures to *happen infinitesimally before* others that have already been processed.

Consider a sequentially executed simulation; there is no ability to roll-back events should a new one be generated that *happens before* one previously processed – if a mechanism

Table 7.1: Events and their virtual times and tie-breaking values for the example shown in Figure 7.2.

Event	(time, tie-breaker)
A	$(1, 0.1)$
B	$(1, 0.15)$
A'	$(1, 0.40)$
B'	$(1, 0.30)$
A''	$(1, 0.20)$

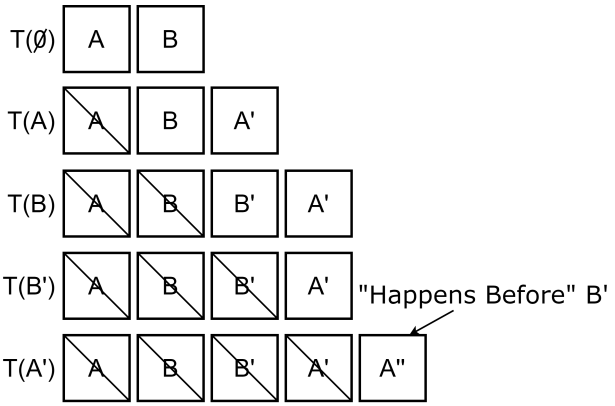


Figure 7.2: Diagram showing event processing queue state while processing previously given simultaneous events. $T(-)$ represents the time in the simulation after a given event has been processed. Event A'' is created by A' but according to its tie-breaker value should happen before B' which has already been processed as designated by a slash.

cannot be used for generating a valid ordering in sequential, then it is unlikely to consistently work for parallel executions either. As an example, two events, A and B each generate a zero-offset child event: A' and B'. A' generates a single zero-offset child, A''. The events are each scheduled to occur at the same point in virtual time but have uniform random generated tie-breaking values to determine the order of events not directly caused by each other; the virtual time and tie-breaker values for these events are shown in Table 7.1.

In this sequential simulation, we would start by processing events A and then B. This would schedule two new zero-offset events A', and B'. We could process these two in order according to their tie-breaker values without issue (B' first). However, after processing A', another child event is created, with a tie-breaker value that is lower than B'. The simulation processing queue state for this example can be observed in Figure 7.2. The result of this is an irreparable error in the simulation. In a sequential execution, an effective negative-time event was received, breaking Lemma 3. In an optimistic execution, we will enter an infinite

rollback scenario and cause instability in sequential.

This is exactly the challenge of creating a fair deterministic ordering of zero-offset events: it is not possible for the simulator to know if a zero-offset event will create another zero-offset event until after its processed. Thus, how can a fair random ordering of zero-offset tie events be determined?

7.4.2 Biased Random Causal Ordering

It is now clear that a single pair of randomly defined tie-breaking values is insufficient for ordering two events given the existence of zero-offset events. It is always possible to violate causality, and that should *never* be possible; any scheme to determine ordering two events must guarantee causality is maintained.

We know, however, that a simulator that utilizes standard virtual time can generate a deterministic total ordering of events if no ties exist because all events have a unique time. Thus, the final ordering should just be sorted, monotonically increasingly, by their timestamp. This is possible because when new events are created, they are encoded with an offset: an additive 'time from now' value. So long as that value is not negative, it guarantees causality is maintained; as long as the value is positive, it guarantees determinism.

What if, when generating a zero-offset event, instead of using a random tie-breaking value from (0,1), we generate that random value but *add* it to the random tie-breaking value of the parent event. The new tie-breaking value will be strictly greater than the parent, assuring that any comparison of the two will always order the parent first. A LP-Event diagram using the same generated values in the previous example using Table 7.1 but adding consecutively generated tie-breaking values when creating child events is shown in Figure 7.3.

Proposition 2. *Providing, in addition to the standard virtual timestamp, a secondary value defined by summing deterministic uniform random values generated by causally related zero-offset events and comparing tied events based on that will yield a randomized-but-biased deterministic ordering of events, including zero-offset events, without violating causality.*

Proof. Given four simultaneous events A, B, C, and D where $A \rightarrow B$ and $A \rightarrow C \rightarrow D$ and B, C, D are zero-offset from A with additive tie-breaker values. Assume that an event D is an event that violates Lemma 3 in conflict with a previously processed event B. The tie-breaker value of D is strictly greater than that of its parent: C. The only way for D

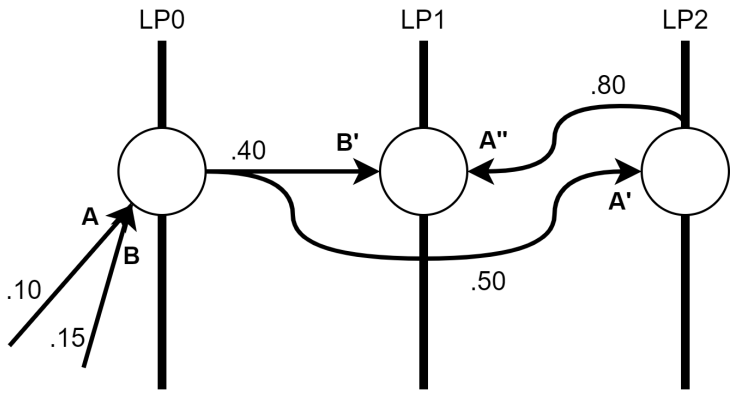


Figure 7.3: LP-Event Diagram depicting events all simultaneously occurring with the same virtual timestamp but depicted with varying legal tie-breaking values encoded using the additive scheme described in Section 7.4.2. The final ordering in the simulation for these tied events is: $[A, B, B', A', A'']$**.**

to be created so that it happens before B is if C also happened before B. If C happened before B, then no ordering of B and D would violate Lemma 3. Thus, by contradiction, additive tie-breaking values will safely allow for zero-offset event tie-breaking without violating causality established from Lemma 3. □

This does allow for randomness to be introduced in the ordering of events but it introduces some potentially unintended bias into the comparison of tie-breaking values. For example, given two events scheduled for the same time: A and B. Event A creates a zero offset child A' who creates a zero offset child A''. The four events are each scheduled for the same virtual time. If the tie-breaker value for descendants is additively generated, then the probability that event A'' comes before event B is not 50%. It is possible that, based on the tie-breaking value, that A'' happens before B, but it is far more likely that B happens before A'' because the expectation for its tie-breaker value is lower than that of A''.

This is because the tie-breaking value encoded into events A' and A'' are no longer a uniform random value from (0,1) but is instead sampled from an Irwin-Hall distribution [81]. The expected value of an Irwin-Hall value is proportional to the number of uniform random values added to generate it, and thus the more zero-offset descendants recursively created, the less likely they are to happen before another, independent, event. In a way, this may seem natural, but it is important to remember that these events are all scheduled by the model to happen at the same time.

This method will assuredly result in a deterministic ordering, and this ordering will be somewhat randomized based on the simulation's input seed. Is there, however, a fairer way to break ties within zero-offset events?

7.4.3 Unbiased Random Causal Ordering

When trying to find a fair way to order simultaneous events, something to consider is whether it even makes sense for other events to interject between an event and its zero-offset child. The two related events are supposed to occur simultaneously, with the exception of one being causally dependent by the other (the child *happens infinitesimally afterward*). Our next proposition is, thus, based on the following assumption:

Assumption 6. *The only event that can interject between a parent event and its zero-offset child is another, sibling, zero-offset child from said parent event.*

What made the additive tie-breaker value in Section 7.4.2 successful in establishing a deterministic total ordering was that it guaranteed that a zero-offset child event will have a value strictly greater than its parent. It also established a unique timestamp that will not – should there be more descendants – violate some established order of other already processed events.

We can devise another mechanism that will ensure that any subsequent zero-offset children of a parent event be processed before other independent events (with a greater tie-breaking value than said parent). Consider ordering two sequences of numbers. There are numerous ways to pick some relation and choose which should be considered first in an ordering of the two sets. One common way is through *lexicographical ordering*. Just as one would find where to place new words into an English dictionary, we start by comparing the first items in the two sets. Should those two match, we recursively compare each of the subsequent items in the sequences until we find a pair that is different. Determining which of the two is 'less' than the other, lexicographically, is then based on that final comparison.

For example, let us consider two sequences $S_1 = [5, 3, 9, 3]$ and $S_2 = [5, 3, 4, 6]$. Lexicographically, $S_2 < S_1$ because in the third component-wise comparison, $4 < 9$. We also observe that it does not matter how long the sequence of S_2 is; as long as those first three numbers remain in each sequence, then $S_2 < S_1$ *always*.

We can extend our definition of virtual time again to include, not a single tie-breaking value, but a *sequence* of tie-breaking values. The sequence is comprised of the tie-breaking

values of historical zero-offset parental events – whenever a new event is created, a new tie-breaking value is generated and appended to the back of this running list. When a regular-offset event is created, the sequence is discarded, and then the newly generated tie-breaking value for the current event is added as the sole value.

When comparing two tie-breaking sequences of different lengths but all N components of the shorter one match the first N components of the longer sequence, priority is given to the shorter one as this can only happen if the event owning the shorter sequence *caused* the other. Referring back to the English dictionary analogy, the single-letter word 'A' comes before 'aardvark' but 'aardvark' comes before 'apple'.

The result of this extension allows for us to make every single event in the simulation comparable to one another via the new virtual time signature. In this case, the virtual time signature is defined as the sequence of the virtual timestamp followed by the regular-offset tie-breaker and all consecutively generated zero-offset tie-breakers. Because of the benefits of lexicographical ordering, causality is guaranteed as any zero-offset descendant's time signature will be strictly greater than its parent, grandparent, etc. The happens-before comparison of any two tied events that are unrelated will, as in Proposition 1, be based on a comparison of two uniform-random values.

Additionally, the comparison of any two tied events that *are* related will also be based on a comparison of two-uniform random values *unless* they are causally locked to a single relative ordering because one is a descendant of the other, yielding:

Proposition 3. *Providing, in addition to the standard virtual timestamp, a secondary sequence of values defined by deterministic uniform random values generated by causally related zero-offset events and comparing tied events lexicographically based on that will yield an unbiased random deterministic ordering of events, including zero-offset events, without violating causality.*

Proof. The exact same logic as demonstrated in the proof of Proposition 2 can be applied here to guarantee determinism as lexicographical ordering provides the same strict less-than/greater-than relation between two events. Using the same events from that example, however, we can observe that should C happen before B, then D assuredly happens before B as it *happens infinitesimally after* C. The determination of the ordering of D and B still falls down to a fair comparison of two uniform random values: the two generated for B and

C. The probability of D happening before B is, effectively, randomly determined by a fair coin flip and is unbiased. □

We recognize that the restriction imposed by Assumption 6 detracts from the generality of Proposition 3, and relaxing this assumption is a point of continuing research.

7.5 Experimental Models

We evaluate our proposed solution with various methods to observe its capability as well as the performance impact of the solution's overhead. We first analyze its performance impact on a standard PDES benchmarking model: `PHOLD` with a strong-scaling study. In this model, event ties are infrequent and there are no zero-offset events.

None of the results generated from running `PHOLD`, however, will tell us whether the ordering of events was consistent from run to run. To do that, we developed a new PDES model which intentionally creates as many event ties as possible with and without zero-offset events, and the final LP state is dependent on the ordering of these events. We show the performance impact in a strong scaling study using this new model.

We also stress test the simulation with an exceedingly high number of event ties in a worst-case type scenario and observe the performance impact in a third strong-scaling study based on a variation of that second model.

All experiments presented below were performed on the ROSS PDES engine with and without the unbiased tie-breaking mechanism described in Section 7.4.3.

7.5.1 `PHOLD` Model

We utilize the PHOLD benchmarking model included with ROSS. It is a generally well-known tool for analyzing the performance of PDES systems [82], [83]. It generates a lot of events that are sent between an LP's self and other LPs based on a configurable probability parameter.

Because of the sheer volume of events that this model can generate at large scales, this model is not immune to the possibility of event ties but is not exceedingly likely because new events are scheduled at some point strictly in the future at a time that is the result of an exponentially random number valued offset.

7.5.2 `Event-Ties` Model

Given any implemented solution for the problems discussed in this work, it is difficult to know for certain that the determinism observed in a couple of small-scale experiments is true determinism or coincidence. This is especially the case if event ties are relatively infrequent.

Algorithm 2 Event-Ties: Behavior of LP P on receipt of event e

Given:
r: preset threshold for desired remote events
l: preset length of zero-offset event chain

P.mean_val$\leftarrow Mean$(P.mean_val, e.val)

Create new event n
n.val$\leftarrow$ Random Integer [0,100]
if Random_Unif() $< r$ **then**
    `dest`$\leftarrow$random, non-self, LP
else
    `dest`$\leftarrow$self
if e is the l^{th} zero-offset event **then**
    Send n to arrive at `dest` at time `P.now()`+1 //regular-offset
else
    Send n to arrive at `dest` at time `P.now()` //zero-offset

To validate that our solution accomplishes its core goal, we developed a ROSS model specifically designed to create a scenario where exact event ordering is absolutely critical to deterministic final results. A model that utilizes a running sequence of mathematical computations with strict ordering, or non-commutative operations, operating on local LP state will make non-determinism evident. Two identically configured simulations with even slightly different event ordering will yield a different final mathematical result.

For our model, we leverage the non-commutative property of the mean of means:

$$Mean(Mean(A, B), C) \neq Mean(Mean(A, C), B)$$

To utilize this property, when an event is received by an LP, its state is updated to be the mean of its current state and the encoded state of the event. If two events are received by an LP at a simultaneous point in virtual time and a deterministic rule for simultaneous event ordering is not implemented, then because of Assumption 1 and the above property, simulation determinism is not guaranteed.

The core behavior of the model is described in Procedure 2. When a new event is received, the recursive mean value in LP state is updated and a new event with a new random integer is created. Where this event is destined is based on a probability challenge so that the number of remote simulation events can be controlled. When this event is to be scheduled is determined by where in the zero-offset chain this event is. If some configured number $l - 1$ same-virtual-time events causally preceded this one, then send the new event with an offset of one, thereby breaking the zero-offset chain. If this event *is not* the l^{th} event in the zero-offset chain, then create another zero-offset event.

7.5.3 `Event-Ties-Stress` Model

In Section 7.5.2, we showcased an example model that would generate a large number of tied events with and without zero-offset timings. But we can make this even harder to showcase the capabilities of our tie-breaking mechanism.

Wherein Procedure 2 creates simultaneous chains of zero-offset events, we can instead generate simultaneous trees of zero-offset events shown in Procedure 3. The difficulty of this model can be scaled by adjusting the height and degree of each generated zero-offset tree. The lexicographical ordering operating on the tie-breaking sequences is particularly useful in this case as it is able to quickly determine which event happens before another, regardless of if one event is a zero-offset offspring, sibling, cousin, nephew/niece, or unrelated to the other. Furthermore, we have shown in Section 7.4.3 that the probability of any two events being ordered a specific way in the simulation is 50% unless one is a zero-offset descendant (child, grandchild, etc.) of the other. In that case, there is a single possible relative ordering.

Also shown in Procedure 3 is a mechanism to make sure that only one event from a zero-offset tree creates a new zero-offset tree. Without this, there would be exponential growth of events as the simulation proceeds making it progressively more difficult to complete.

7.6 Results

All simulations performed in this work were executed using at most four nodes of the AiMOS supercomputer at Rensselaer Polytechnic Institute's Center for Computational Innovations [5]. Each node contains two IBM Power 9 processors with 20 cores each with clocks rated at 3.15GHz and 512GiB RAM. For simplicity of scaling experiments, we utilized a maximum of 32 processors per node. Inter-node communication is facilitated by an Infiniband

Algorithm 3 Event-Ties-Stress: Behavior of LP P on receipt of event e

Given:
r: preset threshold for desired remote events
h: preset height of zero-offset event tree
c: preset number of zero-offset children generated per event

P.mean_val← $Mean$(P.mean_val, e.val)

for $i = [0 : c)$ **do**
  Create new event n
  n.val← Random Integer [0,100]
  if Random_Unif() $< r$ **then**
    `dest`←random, non-self, LP
  else
    `dest`←self
  n.descendant_sum += i
  Send n to arrive at `dest` at time `P.now()` //zero-offset
if e is in the h^{th} level of the tree & e.descendant_sum is 0 **then**
  n.descendant_sum = 0
  Send n to arrive at `dest` at time `P.now()`+1 //regular-offset

EDR Fat Tree communication network.

Binaries were compiled using IBM's XLC_r compiler and Spectrum MPI.

`PHOLD`: **Strong-Scaling** We wanted to benchmark the performance of this extension of ROSS in direct comparison with the old version of ROSS (without the deterministic tie-breaker). The `PHOLD` model is commonly used to benchmark ROSS (and other PDES systems) and is included in the simulator's codebase [82], [83]. These simulations were configured to each have 2 million LPs with a target of 10% remote events generating a total of 1.3 billion net events.

What we observe in Figure 7.4 is that the deterministic tie-breaker version runs a bit slower at lower numbers of ranks but catches up as more ranks are added. The overhead of implementing the tie-breaker does not significantly impact parallel performance, especially with larger numbers of ranks, and because there are so few event ties, the benefit may not be obvious based on runtime alone.

`Event-Ties`: **Strong-Scaling** Using the `Event-Ties` model, we create a situation with as many event ties as possible. All event timestamps are either zero-offset or integer incremented.

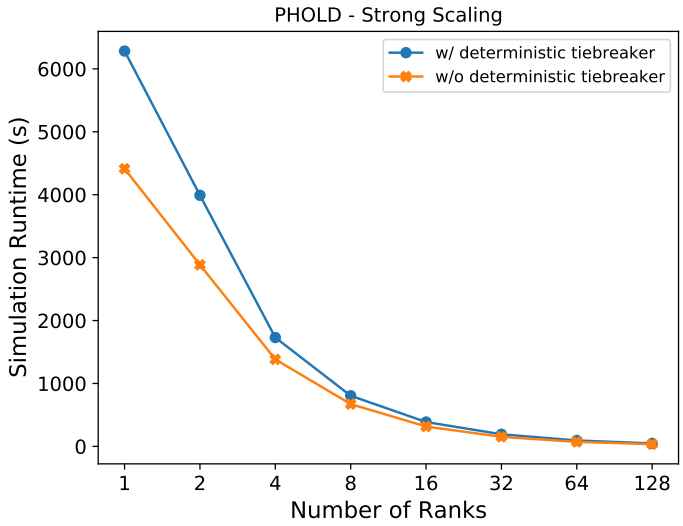


Figure 7.4: PHOLD strong scaling study with and without a deterministic tie-breaking mechanism.

65,536 LPs were simulated, each creating a single regular-offset event which causes a single zero-offset event. In total, this created 39 million events; no event was created that was not tied with another event in the simulation.

We wanted to be able to show in a plot the capabilities of the deterministic tie-breaker version of ROSS in contrast to that of the old version. The deterministic tie-breaker version excels when faced with many zero-offset events, but the old version will give the appearance of stalling as it struggles to make progress.

The reason for this behavior is that when the original ROSS instigates a rollback, it rolls back *all* events with the same timestamp as the target event – it has no way to make two events with the same timestamp comparable. This means that if a single straggler event arrives – which it most likely will – then any progress since the last regular offset event must be rolled back.

Because of this limitation, the only way we were able to show these two ROSS versions in a single plot with this model was to only create zero-offset chains of length two and to instead increase the target remote events to 50%.

In Figure 7.5, we can see the result of this behavior. The old ROSS version appears to do well to a point, and then as more and more ranks are added, the probability of a rollback increases and eventually, it takes considerable amounts of time for the simulation to complete.

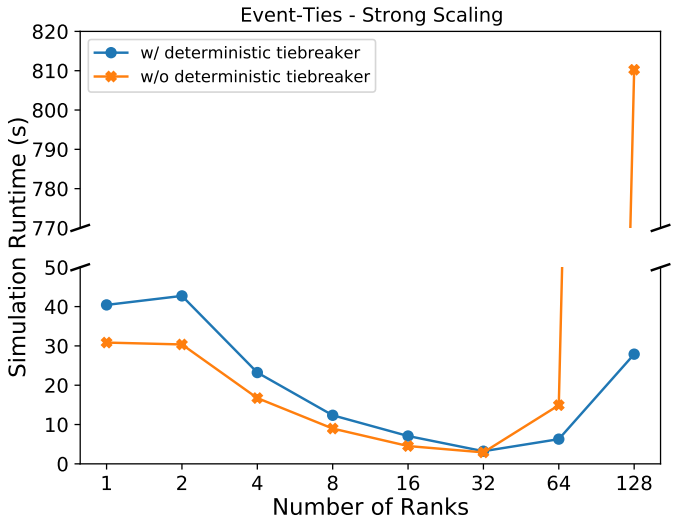


Figure 7.5: Event-Ties strong scaling study with and without a deterministic tie-breaking mechanism. Note that the y-axis is split to account for the last data point.

`Event-Ties-Stress` **Strong-Scaling** This model created zero-offset event trees instead of chains as with the standard `Event-Ties` model. This creates more zero-offset 'siblings'where the tie-breaker sequence must be fully utilized to establish a deterministic ordering.

This model was tested with 131,072 LPs, each generating a 3-ary zero-offset tree of depth five at every integer time-step in the simulation. This created 3.9 billion total events, all with an extreme degree of ties with other events and a target remote percentage of 10%. The small increase in runtime for the $k = 2$ ranks run is because, for this particular simulation model and configuration, the benefit of the additional processing power is not yet able to outweigh the cost of rollbacks which are not observed in the sequential execution.

7.6.1 The Case for Determinism

All simulations in this work utilizing the deterministic tie-breaking feature were verified to exhibit the expected determinism. In contrast, simulations executed without utilizing the tie-breaking mechanism did not consistently generate a reproducible result.

By looking at the results of the `PHOLD` scaling, one might argue that the overhead and performance impact is too great to warrant utilizing the feature. `PHOLD` LP state does not change throughout the simulation, and the generation of future events does not depend on the contents of events received. Thus, even though event ties do occasionally occur in the

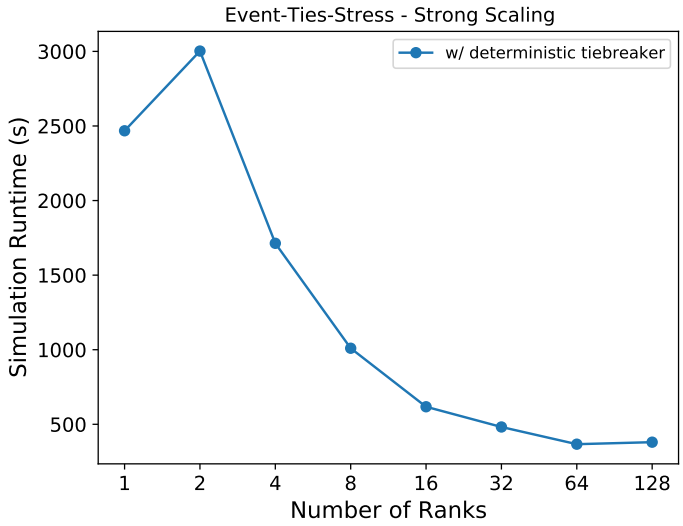


Figure 7.6: Event-Ties-Stress strong scaling study with deterministic tie-breaking mechanism. The old version of ROSS did not tolerate this model well and could not make noticeable progress in any execution mode but sequential.

simulated model, their ordering does not affect how many net events there are in the final simulation.

The same cannot be said for models sensitive to simultaneous event ordering. Table 7.2 shows extracted results from a few runs from the `Event-Ties` scaling results featured in Figure 7.5. We observe small differences in the final results of LP state when there is no arbitration of simultaneous events – a lack of determinism. With the deterministic tie-breaker functionality enabled, however, determinism is restored, and the parallel executions are identical to their sequential counterpart. The difference between the sequential runs with

Table 7.2: Subset of final results of the sequential $k = 1$, optimistic $k = 32$ and $k = 128$ runs for the `Event-Ties` model with and without the deterministic tie-breaker (TB).

Simulation	Net Events	LP1 Final Value
w/o TB: $k = 1$	39,190,528	35.0814
w/o TB: $k = 32$	39,190,528	34.4111
w/o TB: $k = 128$	39,190,528	34.4010
w/ TB: $k = 1$	39,190,528	34.7264
w/ TB: $k = 32$	39,190,528	34.7264
w/ TB: $k = 128$	39,190,528	34.7264

and without the tie-breaker is to be expected and is because the rules dictating order in the tie-breaker simulations also affect the order of events in the sequential version.

With the `Event-Ties` model – and *any* model whose generated events depend on LP state at time of processing – however, event ordering explicitly affects the final result. In the example shown in Table 7.2, it may appear that the executions without the tie-breaker were *close enough*, but this is a product of the simplicity of the model and not indicative of what one could expect with other, more complicated, models. Table 7.2 also shows that simply looking at the number of net events alone is not sufficient for identifying determinism in a simulated model.

If, instead, we couple the remote destination decision in Procedure 2 to be dependent on its current LP state value, then drastically different results are observed. It makes sense that, should LP state play a role in how events are created (if they are created in the first place), then that model will be particularly sensitive to the ordering of simultaneous events. It is important to keep in mind, then – given a standard virtual time simulation – the level of performance observed in a model with simultaneous events may simply be borrowed with determinism offered as collateral.

Lastly, a deterministic result doesn't necessarily imply that it's absolutely the only possible result of a particularly configured simulation. Varying the seeds used to initialize the tie-breaking pseudo-random number generators will allow for the exploration of different possible event orderings deterministically. Furthermore, by removing bias from the breaking of event-ties, we can run many independently seeded executions of the same simulation and make statistical inferences of the generated simulated model.

The goal of parallel discrete event simulation is to recreate a semantically identical sequential simulation but distributed across different processing elements. If, from the general simulation engine's point of view, determinism in the parallel execution is not assured, then it has, by definition, failed in this objective.

7.7 Conclusion

In summary, we have proposed three solutions to the problem of maintaining a deterministic ordering of events in a parallel conservative or optimistically executed discrete event simulation, given the possibility of n-way simultaneous events with and without zero-offset events. This solution leverages a deterministic pseudo-random number generator to encode

values for use in breaking these n-way event ties into a specific, execution-consistent, ordering. This ordering can be unbiased and varied by changing the initial seed of the simulation's random number generators, which allows for further statistical analysis of execution results.

We evaluated the performance impact of the proposed solution and observed a marginal impact in a standard PDES benchmarking model. In the near worst-case, stress-test, benchmark models, the performance of the simulation without the tie-breaking functionality was weakened if it was able to make effective progress at all. Furthermore, a simulation without a tie-breaking feature cannot guarantee that it will deterministically produce an optimistic execution that is semantically identical to the sequential counterpart.

Developers and modelers should always be concerned if a program they wrote generates different results in repeated executions. Did they make a mistake somewhere? Is there a race condition they had not accounted for? We believe that determinism in parallel discrete event simulation is an important tenet and should not be cast aside in the aim of faster performance as any amount of non-determinism is, by definition, unexpected behavior.

CHAPTER 8
SIMULTANEOUS EVENTS IN THE SIMULATION OF HPC INTERCONNECT NETWORKS

CODES' simulation of interconnection communication networks models high-level behavior on the scale of milliseconds based on the actions and decisions made on the scale of nanoseconds in virtual time. To have confidence in the simulated model and the results that it generates, it's important to ensure that consecutive executions of the same simulation produce identical results. This deterministic result is coupled with event processing order.

Typically, virtual time [28] solves this problem, ensuring that events are generally processed in the order that they occur in virtual time. But as discussed in Chapter 7, it is not as simple to accomplish when simultaneous events are allowed. It is even harder when zero-offset event ties are also introduced into the simulation.

CODES simulations try to achieve deterministic ordering by adding small random noise to the virtual timestamps of events. This is somewhat successful, but due to limitations of floating-point timestamps, eventually the precision with which the simulation can increment virtual time is reduced, making event ties more frequent. Additionally, because CODES' final results are based on measurements on event timestamps, this strictly-positive noise influences the final result, making an observation of ground truth that much more challenging.

8.1 Introduction

In Chapter 7, we showed the importance of maintaining a deterministic ordering of events in a parallel discrete event simulation. The final results, and any inferences made from them, are coupled with the final event ordering. If a model's execution isn't deterministic, how can a researcher ever be certain that they haven't written their reverse computation methods incorrectly? Then, even after guaranteeing a particular ordering is deterministically reproducible, how can they be certain that the collected data is that of exemplary model behavior and not an outlier resulting from a particularly special ordering?

Portions of this chapter previously appeared as: N. McGlohon and C. D. Carothers, "Unbiased deterministic total ordering of parallel simulations with simultaneous events,", May 2021, arXiv:2105.00069v1 [cs.DC].

Portions of this chapter are to appear in: N. McGlohon and C. D. Carothers, "Toward unbiased deterministic total ordering of parallel simulations with simultaneous events," in *Proc. 2021 Winter Simul. Conf.*

While chapter 7 itself is focused solely on ROSS, the proposed solutions can have a significant impact on CODES and its models. CODES model LPs chain together various behaviors by sending self events with some near-zero jitter added in to avoid event ties. Because of this small gap in virtual time between two consecutive operations, there is a non-zero chance that another event could arrive in between them and change LP state. Because of this possibility, CODES models must account for this state change.

For example, a terminal LP generates a packet and sends a PDES event to itself to wake itself up in a few virtual nanoseconds to send that generated packet to an attached switch. In this example, a management event interjects between the generation event and the packet forwarding event, changing the terminal LP state to one that prohibits packet injection for a short period of time. When the previously scheduled 'send' event arrives to wake the terminal up to inject the generated packet, it has to check and make sure that the *current* LP state doesn't prohibit injection before proceeding with its originally intended behavior. To avoid this possibility of interjection, the developer uses a zero-offset instead. Thinking that this ensures that the event can't be preempted by another anymore, they believe that they are safe until they find that they no longer have deterministic results.

Another event was scheduled and tied with that zero-offset event (and its parent). The ordering of these events is based on inter-process message arrival times, which is inherently non-deterministic. This may not be immediately obvious to a model developer; they could spend considerable amounts of time analyzing their model and reverse computation methods trying to find an error to no avail. Over the course of developing the simulations for other chapters of this work, a significant amount of time was spent doing exactly that. It was never possible to be certain that reverse computation was written correctly without determinism - which means that we couldn't be certain that our optimistically executed simulations were actually valid.

In this chapter, we make the case that the methods for breaking event ties described in Chapter 7 (specifically Proposition 3) strengthen the CODES simulator from several perspectives:

1. Reduces difficulty of development
2. Decouples tie-avoidance noise from measurements
3. Allows for the statistical sampling of general model behavior

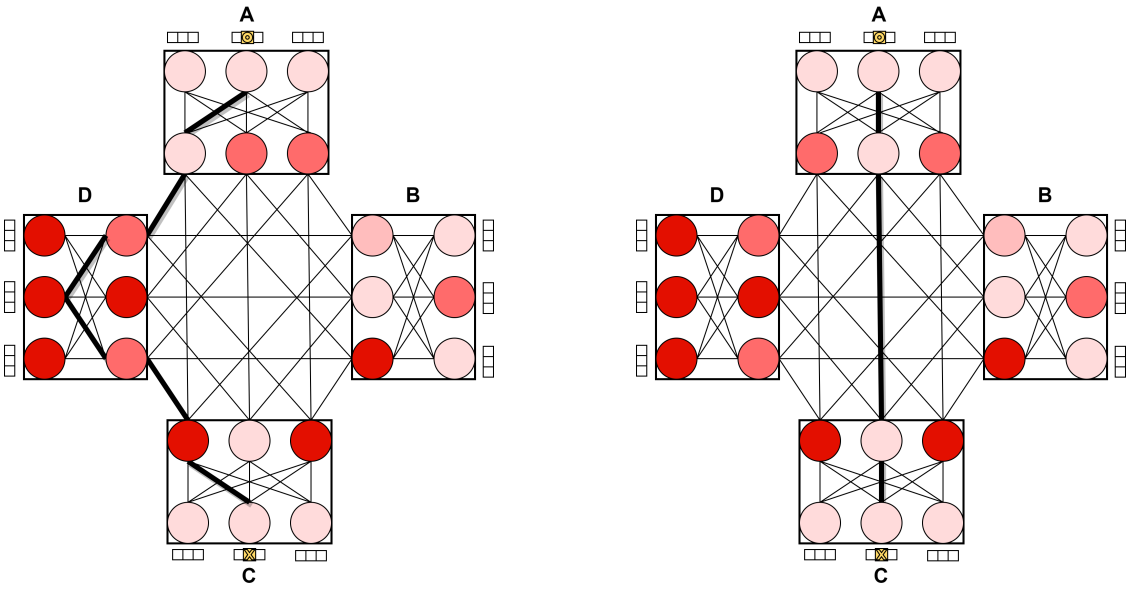


Figure 8.1: A toy example Megafly network. Depicted source terminal in switch group A wants to send a packet to depicted destination terminal in switch group C and chooses either a minimal or non-minimal route based on locally available information. Depending on the system state (and the ordering of past events that resulted in it) different routes may be chosen – such as (a) a poor, non-minimal choice or (b) a preferred, minimal route – and expected latency of the routed packet will vary significantly.

8.2 Background

In PDES, the final results of a simulation often depend on the specific ordering of various related events during execution; CODES is no exception. If simultaneous, in virtual time, events are processed in a different order, that can make the difference on whether or not a particular simulated packet gets forwarded immediately or is stuck waiting in a queue.

While the exact ordering of two packets may seem inconsequential, in a detailed simulation with various complex non-linear mechanisms and forces acting upon the system, there is a number of things that could be triggered by the delay (or lack of delay) of a packet. Packets traveling within an HPC network interconnect can, thus, be considered a system of chaos. Each consecutive decision in the system is the product of every single past decision made up to that point. Akin to the Butterfly Effect (attributed to Lorenz [84]), seemingly inconsequential decisions may have drastic, unpredicted results.

Figure 8.1 shows two examples of possible non-minimal routing in a toy megafly

network. Depicted are visualizations of hypothetical switch usage with increasingly dark colors representing higher levels of usage. A packet en route from the designated endpoint in switch group A to the designated endpoint in switch group C will determine whether it routes minimally or non-minimally based on information locally available on each switch in its path towards the destination. In Figure 8.1a, the packet tries to find the best route available to it and makes the decision to route non-minimally through switch group D, which is highly congested. As a result, the total transit time of the packet could be adversely affected. With a minutely different pattern of switch usage in the source group, however, a minimal route is found and taken with a significantly different total transit time.

It's entirely possible that another packet that could interfere with this first packet was scheduled for the same point in virtual time. The decisions made by the original packet could very well depend on the result of who is processed first. In this case, what seems like an otherwise inconsequential decision yielded significantly different routes traveled.

8.3 Reducing Development Difficulty

CODES simulation models are complex. The codebase itself is defined across over a hundred C/C++files totaling over 87,000 lines of code [85] with many other types of supporting files as well. If determinism is not maintained as new features are introduced, it can be difficult to know where exactly the cause of non-determinism lies. To fortify our confidence in the simulated models, we have made significant efforts to resolve any sources of non-determinism over the course of all other chapters in this work.

This involved, in summation, many days of development time, testing, and analyzing each reverse computation method utilized and addressing any bugs that were discovered. Still, occasionally, a model wouldn't behave deterministically, and more time was spent trying to find the source only for the behavior to temporarily stop without explanation. It was known that event ties could possibly cause non-determinism, but there wasn't any way to avoid them except for adding in random noise to the timestamp. This, however, is a stopgap solution. Due to the limitations of floating-point data fields, as the exponent bits of the value increases (as the number gets larger), the number of bits leftover for the mantissa (the fractional part) is reduced. Adding 32 (or 64 if using doubles) bits of uniform random (0,1) values to this timestamp may lose significant amounts of precision if there aren't enough bits in the original timestamp to represent them in addition to the existing exponent bits. This

means that the very method with which we attempt to avoid event ties will, if the simulation runs long enough in virtual time, fail.

Using the tie-breaking virtual time extensions described in Chapter 7 will mean no more time will be wasted by the developer searching for reverse computations errors that don't exist. If there is non-determinism, then it means that something was implemented incorrectly. If it is deterministic, then it is likely that the reverse computation methods were written correctly. Furthermore, to ensure a simulation is deterministic, any random number generation must be accounted for in reverse computation as well. If the developer no longer needs to generate a random value to add in tie-avoiding noise to their event offsets, it's one less source of random numbers that need to be correctly, exactly, rolled back in reverse computation methods.

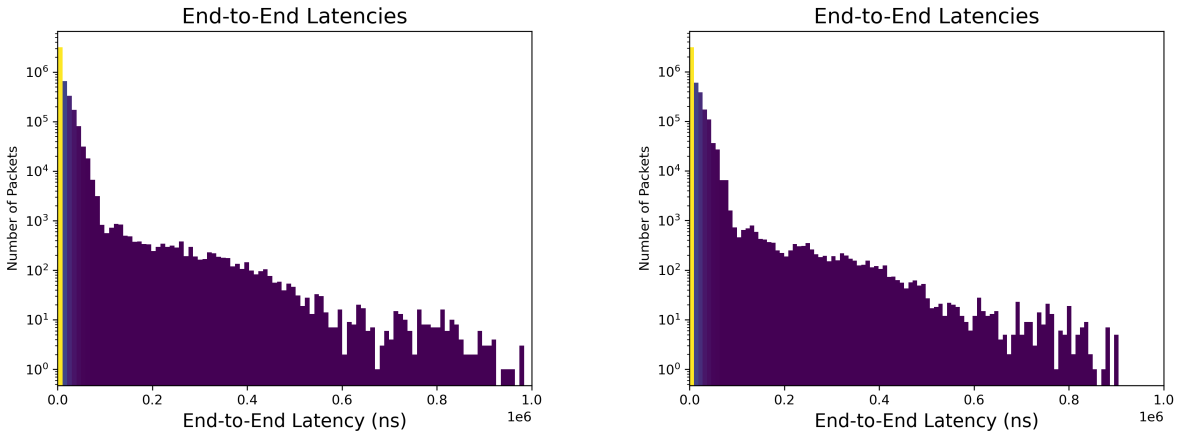
8.4 Impact of User-Defined Noise

Prior to the introduction of the unbiased tiebreaker feature in ROSS, CODES models attempt to avoid non-determinism sourced from tied events by adding small noise to each event. There are a number of issues with this approach.

First, as mentioned in the previous section, it cannot guarantee that tied events won't occur. While ROSS is capable of generating uniform random values with 64-bit precision, which should generally make RNG collision unlikely. In practice, not all of these bits are effectively usable when they are combined with a particular virtual time coordinate which itself is also stored as a 64-bit floating-point value. Thus the likelihood that two events occur at the same point in virtual time becomes more likely as the simulation progresses, and even a single pair of events being ordered differently may result in different final results.

Second, because CODES adds small amounts of additive noise (it isn't possible for subtractive noise as it may result in negative time offset events which are not allowed in PDES), and because CODES makes measurements based on the virtual time of events, the final result becomes directly influenced by this noise. This makes any inference of ground truth difficult to be certain of.

As an example, we've performed a couple of CODES simulations on a 3K node 1D Dragonfly network with a single iteration of the LAMMPS2048 SWM workload and 1000 ranks of random permutation interference traffic. In one simulation, additive uniform random noise in the range (0,1) was added to each network simulation event; in another, no additional



(a) **(b)**

Figure 8.2: End-to-end latencies of two nearly-identical simulations, one with noise added to the offsets of every simulated network event (a) and one without (b). Note the length of the tail of each distribution.

noise was added to any event offsets.

Figure 8.2 shows the impact of this noise on the distribution of end-to-end packet latencies. While the impact on the packet latency distribution may appear small, the length of the tails is important. The maximum packet latency observed when additive noise was added was around 980μs but the maximum packet latency observed without the additive noise was around 904μs: a difference of 80,000ns!

This is significantly more than what we would expect from the small additive noise applied to each event's timestamp. This is because this additive noise being applied to events affects *when* it occurs in the simulation, which directly affects the event ordering. With the additive noise, many events which would have otherwise been tied (which we'll refer to as temporally close) will be slightly offset in a particular order. Without this noise, the ROSS tiebreaker steps in and determines the ordering.

The implications of Figure 8.2 are subtle but of significant importance. With slightly different event orderings, we can observe significantly different packet latency distributions and other metrics. Thus if we were to vary the main simulation RNG seed, we would see different behaviors and orderings. But as long as event ordering is determined by this additive noise, it's impossible to vary the event orderings without also varying the expected random behavior of the model.

Additionally, this additive noise, even if it *could* guarantee that tie events never occurred,

will not result in a fair random ordering of temporally close events. Any noise added to an event will stay added to the timestamps of any subsequent events, which leads to the ordering of two events being based not on the result of two values sampled from a uniform random distribution but from an Irwin-Hall distribution [81] instead.

As an example, picture two packet arrival events that are temporally close: without the additive noise, they would have tied. Let one event represent a packet arriving at its router of origin and the other at the same switch after being routed between a couple of other routers. The first event will have fewer multiples of random noise applied in its lineage, while the other will have had at least a few uniform random noise values applied to the events that facilitated its routing. As a result, it is more likely that the first event will occur *before* the other despite them being temporally close. If we didn't apply this noise and instead relied on the ROSS tiebreaker, the orderings of these two events would be determined fairly.

To be able to effectively and fairly analyze, statistically, the results of two or more simulations with different possible temporally close event orderings, tie-avoiding noise can not be added into the timestamps of events, and a separate tie-breaking mechanism like the one supplied in Chapter 7 must be utilized.

8.5 Sampling General Model Behavior

In the previous section, we observed evidence that small differences in how events are ordered in the simulation can have drastic differences in final simulation output. This is not unique to CODES, but the complexity of the model and the number of binary decisions based on discrete data make it particularly impacted by event ordering. From chaos theory [84], whether something happens or not at time T will directly influence the possibilities of occurrences at time $T + 100$ and the variance of said possibilities could be significant. A simple example is shown in Figures 8.3 and 8.4, depicting two types of events A and B. Event type A will decrement the LP state that it acts on by 1. Event type B will check to see if this LP state is equal to zero. If it is, then it will spawn a new event to do something else. If the LP state is not equal to zero, it will do nothing. The existence of the child event C is strictly dependent on the orderings of the three original events. This child event, C, could itself occur simultaneously with another event in the simulation, and different simulation behavior would be the result of *their* ordering and so on.

To gain a full understanding of general model behavior, different event orderings should

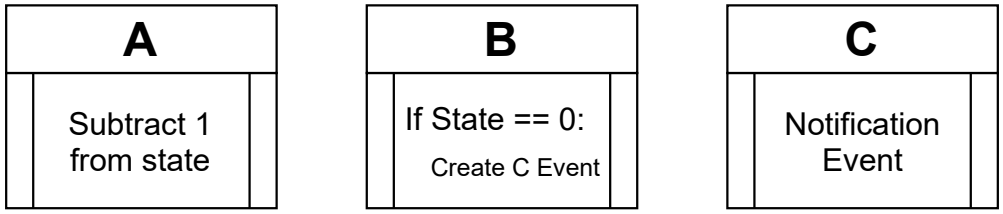
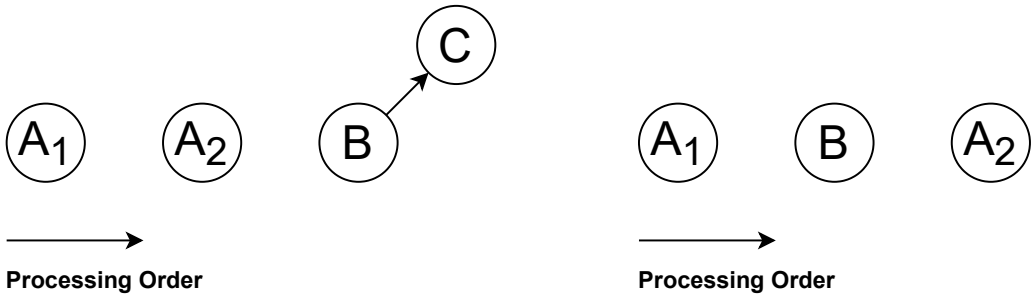


Figure 8.3: A description of three event types to demonstrate the simple impact of simultaneous event orderings. Event type A will subtract 1 from the receiving LP state. Event type B will check if the LP state is equal to zero. If it is, then it will create a notification event C sent elsewhere.



(a) **(b)**

Figure 8.4: Different orderings of simultaneous events A_1, A_2, and B defined in Figure 8.3 will yield different occurrences in the simulation. (a) Given LP state initially set to the value of 2, if the simultaneous events A_1, A_2, and B occur in this order, then B will create a new event C. (b) Given LP state initially set to the value of 2, if the simultaneous events A_1, A_2, and B occur with B happening before A_2, then B won't create a new event C.

be explored. Otherwise, how would one ever truly know if their measurements are statistical outliers from a particularly special ordering of simultaneous events or representative of expected behavior? The author of [86] asserts that the *correct* answer, given N simultaneous events, is the average of all $N!$ possible event orderings. For large models with millions or billions of events (like CODES) with many of them tied, however, the space of all possible event orderings – including any new ones that were previously non-occurring – is unfathomably large.

8.5.1 Varying the Tie-breaking RNG Seeds

By removing all of the additive noise described in Section 8.4 from a CODES simulation, the likelihood of simultaneous events occurring in the simulation greatly increases. Additionally, there are a number of events that are created by CODES that, without the additive noise, become zero-offset events. The high frequency of tied events, particularly with the existence of zero-offset child events, makes the deterministic tiebreaker feature of ROSS necessary to effectively simulate. By using the ROSS tiebreaking mechanism, however, we are able to change the RNG seeds for the tiebreaker feature specifically, leaving the model behavior RNG seeds constant.

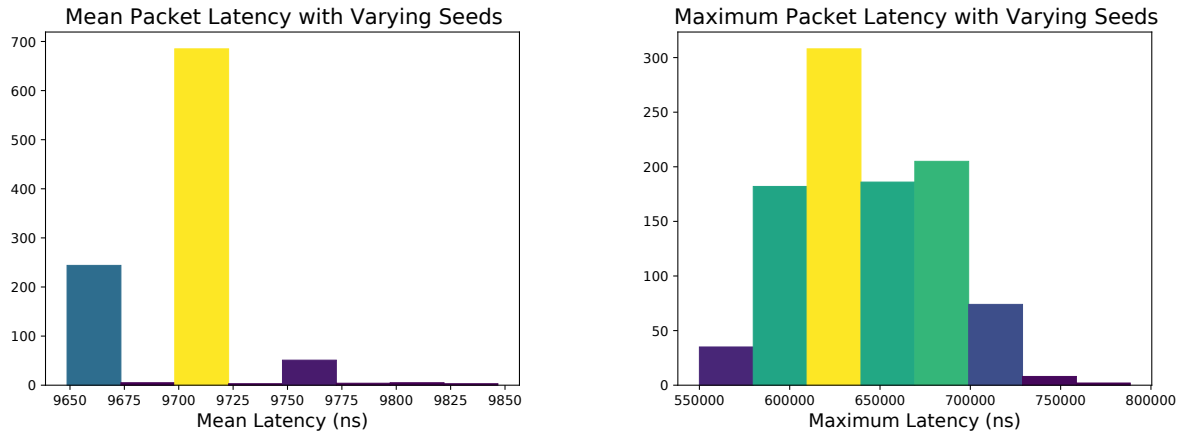
By running many simulations with varying tiebreaker RNG seeds, we are able to sample from the space of all possible simulation event orderings and occurrences. With more simulations, we gain greater confidence in our approximation of the so-called correct answer of model behavior.

To exhibit this, we created an example CODES simulation, the same used in Section 8.4. Specifically, this is a 1D Dragonfly network simulation with 3,178 simulated compute nodes. 2,048 of these nodes are allocated to facilitate a single iteration LAMMPS2048 SWM workload; another 1,000 ranks are allocated to inject synthetic traffic with a random permutation pattern. Each of the synthetic ranks will inject 10 messages of 262144 bytes destined for a random other rank and then pick a new random destination to send 10 more messages. This repeats until the primary workload, LAMMPS2048, completes.

The random permutation workload was specifically chosen as it, by nature, creates "hot spots" of congestion in the network. Adaptive routing will attempt to spread these hot spots around and routers' decisions are based on locally available information. With high degrees of event simultaneity, routing decisions may be dependent on where the event requesting said routing decision falls in a particular event ordering. Thus, we should expect final results to be influenced by different choices of tie-breaking RNG seeds.

8.5.1.1 *Distributions of Final Aggregated Metrics*

Figure 8.5 shows the distribution of the mean and maximum packet latencies found in each of 1000 simulations with varying tie-breaking RNG seeds. This means that different orderings of simultaneous events are explored. We observe that there are final results that are more likely to fit in certain bins than others and that the most common occurrence tends



(a) **(b)**

Figure 8.5: Histograms of the mean (a) and maximum (b) packet latencies over 1000 simulations with different tie-breaking RNG seeds.

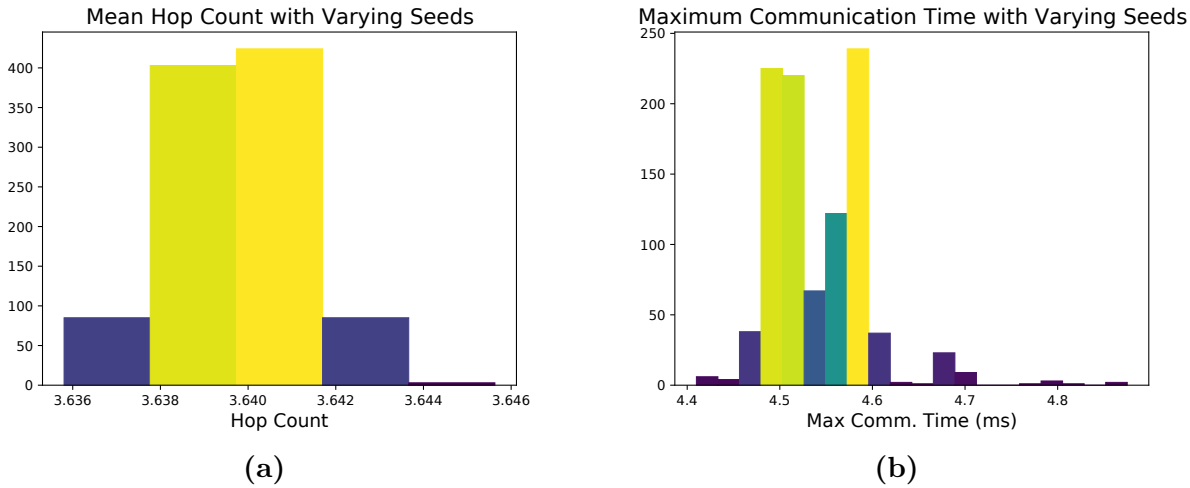
to be close to the mean.

A plot showing the minimum latency across all simulations is not shown as each simulation had at least one packet with a minimal latency of 701.6ns. This minimal time is representative of a packet being injected into a router and being immediately forwarded to a terminal attached to the same origin router. Such a path was required in each of the 1000 simulations and different configured simulations may produce different distributions for minimum packet latencies.

Something of note is found looking at the mean latency distribution: there are a couple of other modes of alternate common behavior. These modes are likely due to the inherent nature of the model and that different configurations and types of simulations explored with a similar study may yield different possible common alternate modes.

We can look at other metrics as well and observe similar behaviors. The distribution of the mean hop count of all packets in each simulation is shown in Figure 8.6a. The limits of the x-axis show us that there is a very narrow spread of possible average hop counts as a result of event ordering. This tells us that the decisions that generally determine route length are not greatly affected by event ordering.

That does not mean, however, that small perturbations in routing decisions can't noticeably affect packet performance. Looking at Figure 8.6b shows the distribution of maximum communication time of the primary LAMMPS2048 workload in each of the 1000 simulations. The spread of this has a range magnitude of ≈500,000ns but has a roughly



(a) **(b)**

Figure 8.6: Histograms of the mean packet route hop counts (a) and primary LAMMPS2048 workload maximum communication times (b) found in each of 1000 simulations with varying tie-breaking RNG seeds.

normal distribution averaged between 4.5 and 4.6ms. While the resulting path lengths, on average, are generally unaffected, the impacts of different routing decisions based on different simultaneous event orderings can be measurable.

8.5.1.2 *Aggregated Distributions of Per-Packet Metrics*

In addition to observing the distribution of final output metrics with varying tie-breaking RNG seeds, we can also combine per-packet metrics across all 1000 simulations. This allows us to see information about how packets facilitating this specific simulation configuration may behave if alternate simultaneous event orderings are explored.

Because capturing per-packet metrics will generate a lot of data from even just a single simulation, capturing *all* per-packet metrics for 1000 simulations is infeasible. To reduce the overall file size of the data, only 1/100th of the total communicated packets were tracked. Capturing the full packet information for all 1000 simulations would have been nearly 2TB of data, making file management and processing difficult. By limiting the number of tracked packets, the corresponding data that the following figures represent was approximately 20GB in size.

Figure 8.7 shows the distribution of the length of the path that each tracked packet took en route to its destination. By combining the results of 1000 simulations with different orderings of simultaneous events, we can be more confident in our estimations of expected

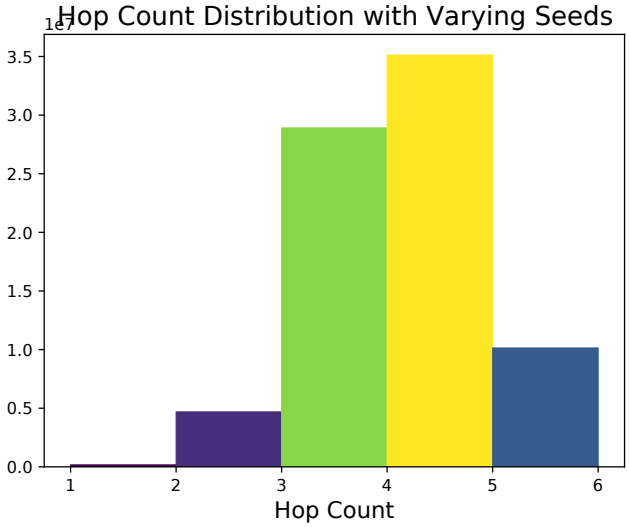


Figure 8.7: Histogram of the path hop counts of all tracked packets accumulated over 1000 simulations with varying tie-breaking RNG seeds.

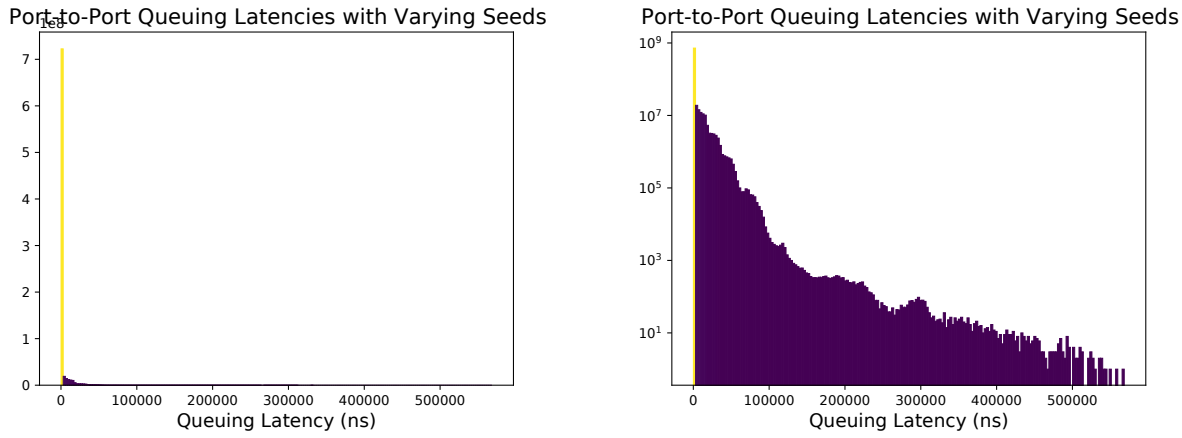
behavior for this metric.

Similarly, we are able to accumulate latency data for each of the tracked packets. Specifically, we observe the port-to-port queueing latencies of packets that they incur as they are routed by each router. The more packets that exist on various buffer channels in routers, the longer that it will take for the routers to be able to transmit packets further back in the queue. Additionally, due to the credit-based flow control, back-pressure may materialize, and these queues will increase in length as long as the routing ability of downstream routers is fully saturated.

Figure 8.8 shows the distribution of tracked packet port-to-port latencies across all 1000 simulations with varying tie-breaking RNG seeds, in linear and logarithmic scale. This is specifically the amount of time from when a packet is received by a router to when it is removed from its output buffer.

What we observe is that an overwhelming majority of packets are quickly transmitted at each router in this configuration. However, we can observe a comparably small but noticeably long tail of outliers. These outliers are relatively infrequent.

It's possible that the queuing latency observed by a packet in a single router may be a rare occurrence, but the accumulation of consecutive port-to-port latencies observed by a packet along its path to its destination will accrue to become an even larger end-to-end latency: the amount of time elapsed between a packets injection to and ejection from the



(a) **(b)**

Figure 8.8: Histogram of the port-to-port queuing latencies of all tracked packets accumulated over 1000 simulations with varying tie-breaking RNG seeds. (a) Linear y-scale. (b) Logarithmic y-scale.

network.

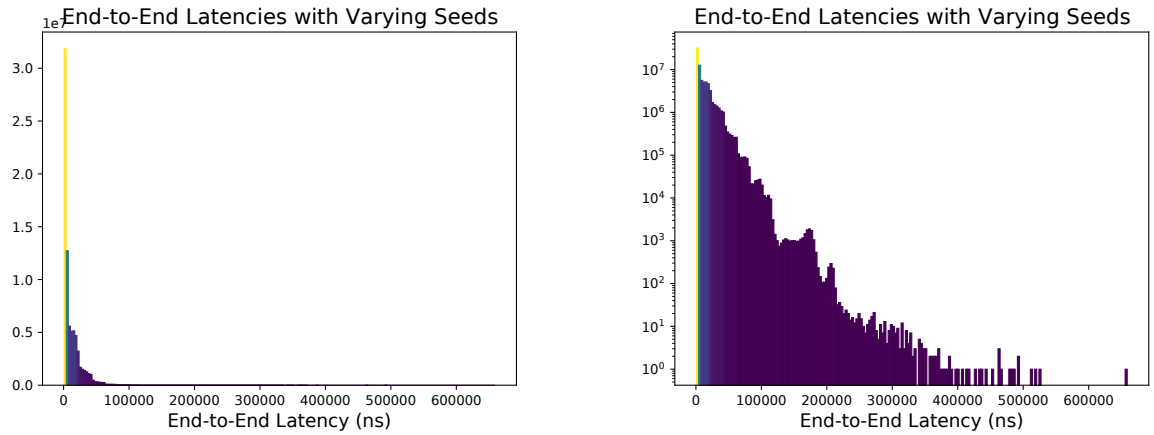
A distribution of the tracked packet end-to-end latencies can be observed with linear and logarithmic scales in Figure 8.9. Similar to Figure 8.8, the vast majority of packets have short end-to-end latencies, but a non-negligible amount of them have longer latencies. Furthermore, the performance of many workloads, specifically those with latency-sensitive collective operations, can be severely affected by even a single outlier packet.

Researchers and system designers may desire to look at such metrics and should not be confined to observing the consequences of a single pre-defined ordering of any simultaneous events. With the fair random ordering proposed in Chapter 7, this is made possible.

8.5.2 Varying the Behavior RNG Seeds

Another approach of simulation analysis could come from varying not only the tie-breaking RNG seeds but also the *LP behavior* RNG seeds. These seeds are what dictate the results of any RNG call made by the LP for behavioral decisions. Progressive adaptive routing [66], for example, determines potential packet routes based on a random sub-sampling of possible minimal and non-minimal routes. These non-minimal routes are also the result of a random decision to determine the intermediate group that a packet must route to before being routed to its original destination.

The results of a single simulation will yield results based on the specific set of decisions



(a) **(b)**

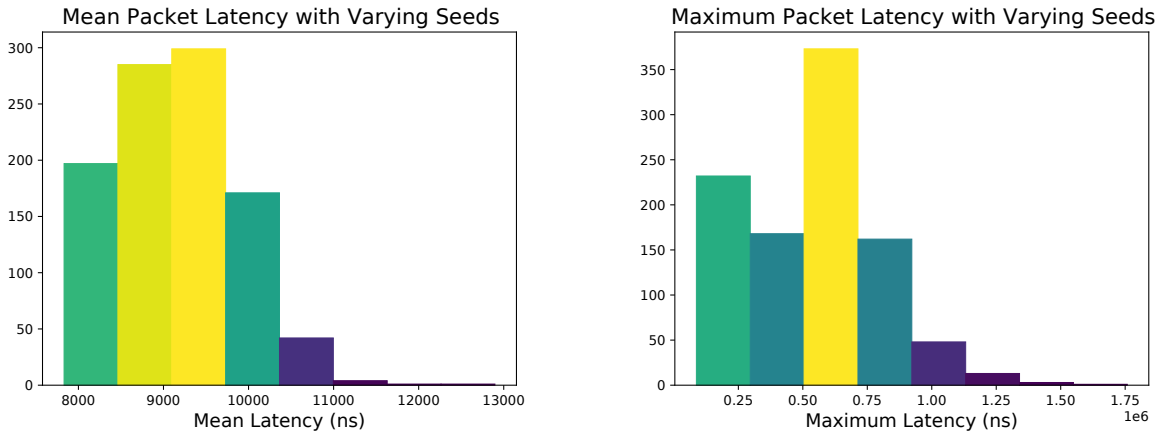
Figure 8.9: Histogram of the end-to-end transit latencies of all tracked packets accumulated over 1000 simulations with varying tie-breaking RNG seeds. (a) Linear y-scale. (b) Logarithmic y-scale.

made by all LPs that generated the simulated event sequence. While it may be possible to make general inferences based on this data – as we have in nearly every study in this work – it should not be accepted, by itself, as the ultimate ground truth of general expected behavior.

The authors of [45] propose that to effectively quantify the performance of a particular network, multiple workload sets and job-to-terminal mappings should be explored. While this is true, this should hold true even within a single configuration. As an example, we can analyze the simulations executed in Section 8.5.1. The workload set used was a combination of the LAMMPS2048 SWM workload and a synthetic random permutation workload. The random permutation workload, by definition, will generate traffic based on RNG calls. The traffic generated by this workload will result in specific hot spots of congestion in different parts of the network.

The decisions made by the random permutation workload, specifically the locality of the specific source-destination pairs for generated messages, and the decisions made by routers, specifically the choice of intermediate adaptive routes, will have a significant impact on the observed performance of the primary workload. This is because different amounts of interference traffic at different parts of the network at different times may – or may not – affect the primary workload, depending on its communication needs at any specific time.

Similar to the previous section, we ran the same simulation as before 1000 times but varying both the tie-breaking RNG *and* the behavioral RNG seeds. This yielded similar but



(a) **(b)**

Figure 8.10: Histograms of the mean (a) and maximum (b) packet latencies over 1000 simulations with different LP behavioral RNG seeds.

unique simulations with different patterns of occurrences. We can compare the final observed results from each simulation as before.

8.5.2.1 *Distributions of Final Aggregated Metrics*

Looking at Figure 8.10, the first thing we should note is that the spread of observed minima and maxima is significantly increased from when we were only varying the tie-breaking RNG. This shows that the ordering of simultaneous events, at least in this model, is not as consequential as the decisions of where and when events occur in the first place.

What this also makes clear is that observing the results of a single behavioral RNG seed could potentially yield measurements that are, in the grand scheme of things, incredibly unlikely to occur.

The same effect is observed in Figure 8.11; specifically, we see a greater variance in the data as compared to Figure 8.6. The shape of the distribution of the maximum communication times across all 1000 simulations is of particular interest. There are a few simulations with high-discord data that is resulting from a particularly unlikely scenario.

Where Section 8.5.1 argued the importance of exploring possible orderings of simultaneous events, we can argue here the importance of exploring other possible outcomes from all randomness existing in the model. If the one simulation seed utilized in a simulation yielded a maximum communication time measurement of 7ms, it could be nearly double the value of the mean observed by all 1000 samples. Inferences made on that single data point could be

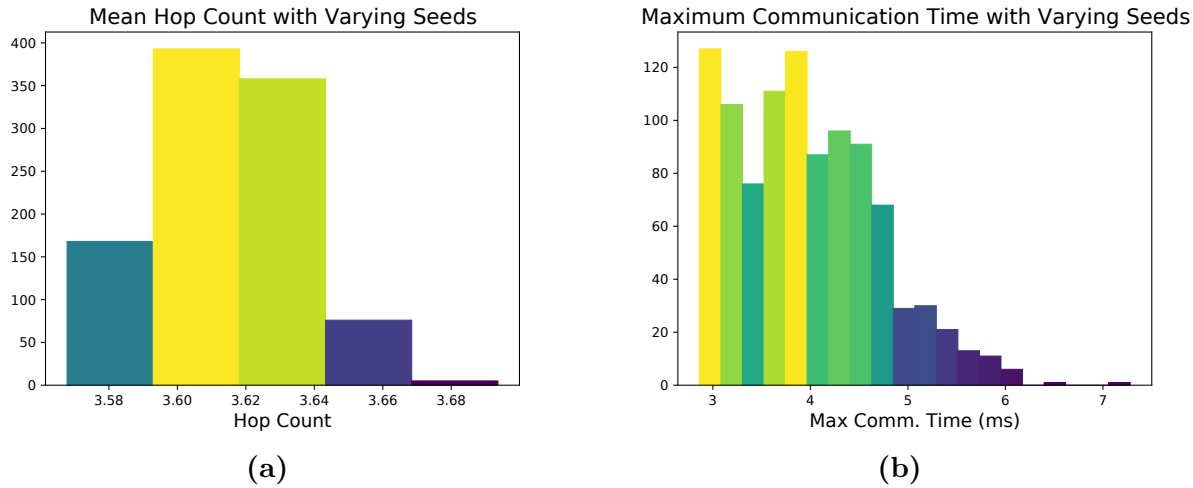


Figure 8.11: Histograms of the mean packet route hop counts (a) and primary LAMMPS2048 workload maximum communication times (b) found in each of 1000 simulations with different LP behavioral RNG seeds.

incorrectly skewed by this comparably rare occurrence, and thus, multiple samples should be performed and measured to increase confidence in the final result.

One might argue that it was always possible to vary the behavioral seeds and generate similar data spreads. While that is somewhat true, it hasn't been statistically sound in the presence of simultaneous events. It was either non-deterministic when it should be (and thus not a valid total ordering of events like its sequential counterpart), or it relied on some inherent bias to order simultaneous events.

This implies that, before the introduction of the fair tie-breaking in Chapter 7, individual seeded simulations were not true samplings of the space of all possible event occurrences. It was sampling an unfair, biased possibility space. With the addition of the tie-breaking mechanisms into ROSS, we can generate histograms to visualize the distributions of possible simulation outcomes and approximate the general expectation of model behavior.

8.5.2.2 *Aggregated Distributions of Per-Packet Metrics*

With the introduction of the fair tie-breaking mechanism, it is then fair to aggregate per-packet data across these 1000 different simulations like we did previously. Figure 8.12 shows a near-identical histogram of the per-packet path hop counts of all tracked packets in all 1000 simulations. This is largely due to the fact that the locations of workload rank to endpoint mappings are kept constant across each of the 1000 simulations. Thus, the mean

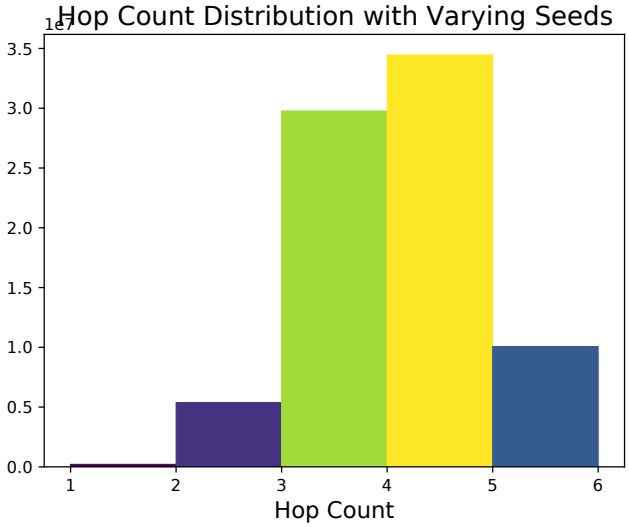


Figure 8.12: Histogram of the path hop counts of all tracked packets accumulated over 1000 simulations with varying LP behavioral RNG seeds.

distance between any two points in the network is equal in all simulations in both cases.

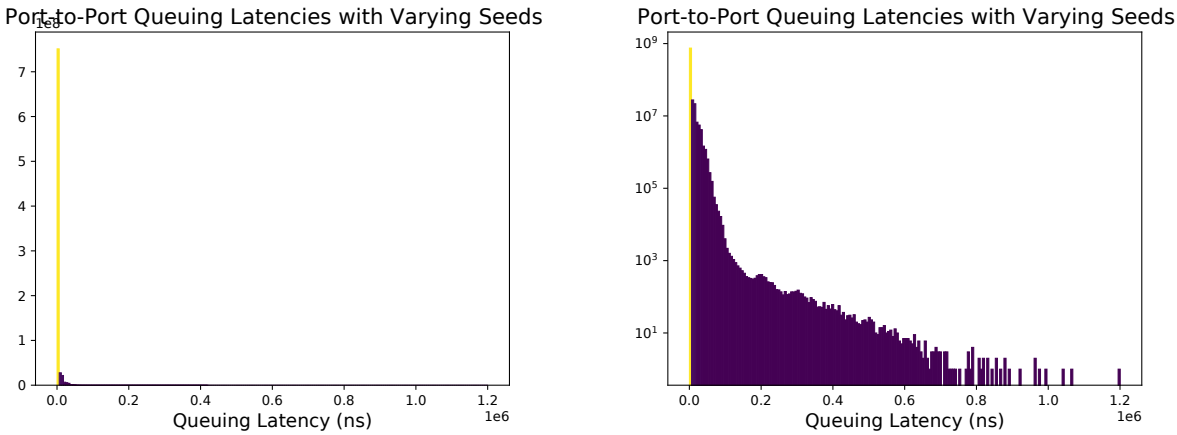
We can also combine the per-packet latency metrics just as we did before. This will give us particularly interesting information in discovering potential weaknesses (or oversights) of the model. For example, in Figures 8.13 and 8.14, we see that the tail of maximum observed port-to-port and end-to-end latencies is even greater than it was before.

Particular randomized workload and routing decisions have led to a maximum port-to-port and end-to-end packet latency of being on the order of *milliseconds* which is an incredibly long amount of time when the general communication time itself is averaging at around four milliseconds.

This could mean that there is a potential flaw in the design of the topology, or the routing, or any congestion avoidance/control mechanism. It could be a trait inherent to the nature of the simulated network itself. Either way, without a true and fair method to sample the space of possibility, there are some occurrences that would have, due to ordering bias, been rendered impossible and never explored.

8.6 Conclusion

In summary, this chapter presented work describing the impact of the fair, deterministic, tie-breaking mechanism presented in Chapter7 on the CODES communication network



(a) **(b)**

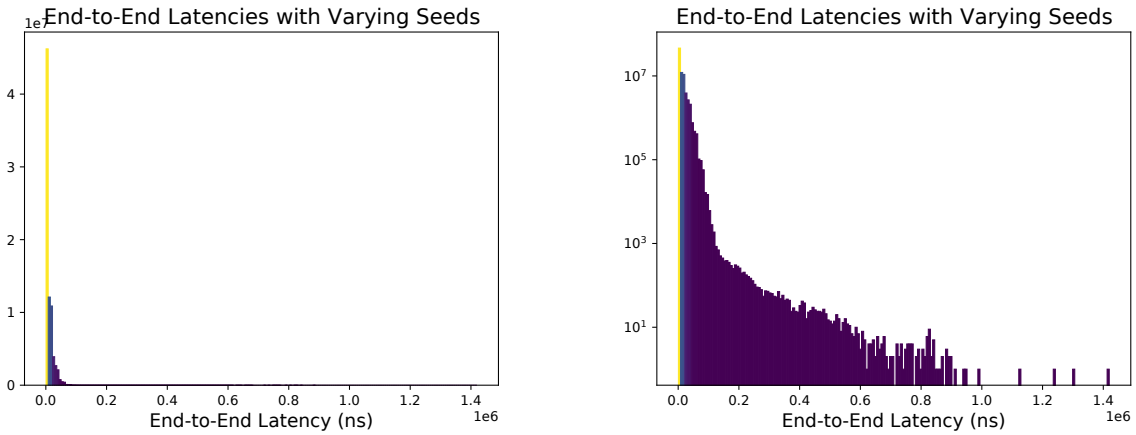
Figure 8.13: Histogram of the port-to-port queuing latencies of all tracked packets accumulated over 1000 simulations with varying LP behavioral RNG seeds. (a) Linear y-scale. (b) Logarithmic y-scale.

interconnect simulation system. We argued that by reducing potential sources of bugs in optimistic-parallel-execution-necessary reverse computation that model development is easier with the tie-breaking mechanism.

Additionally, we demonstrated the impact that user-defined tie-avoidance noise has on final results and argued that it was simply a "close enough" solution but is theoretically unsound as it cannot with certainty prevent the occurrence of tied events. We observed a significant difference in final results with and without the user-defined additive noise applied to event timestamp offsets, implying that the resulting event ordering found from this noise has a substantial impact on final results.

This underlines the importance of exploring other possible event orderings to gain a greater understanding of general model behavior. We presented two sets of simulations, one varying the tie-breaking RNG seeds while leaving model behavioral RNG seeds constant, and another set also varying the behavioral RNGs. Where varying the behavioral RNG seeds will generate different unique occurrences in the simulation, varying the tie-breaker RNGs is essentially observing the consequence of different orderings of any simultaneous occurrences in said simulation.

We observed that the simulations varying the tie-breaking RNG seeds showed narrowly spread data, implying that there is an impact of event ordering in CODES simulations but not of great significance. When varying the behavioral RNGs as well, we observed significant



(a) **(b)**

Figure 8.14: Histogram of the end-to-end transit latencies of all tracked packets accumulated over 1000 simulations with varying LP behavioral RNG seeds. (a) Linear y-scale. (b) Logarithmic y-scale.

differences in final result data. This shows that the decisions that determine when and where occurrences happen in the simulation are of greater consequence than the orderings of simultaneous events. It is important, however, to explore them both to gain a deeper understanding of all potential model behavior.

PART IV

RELATED WORK AND CONCLUSION

CHAPTER 9
RELATED WORK

9.1 On Reducing Inter-Job Interference and Quality of Service Techniques

There are different approaches to address run-to-run variability on HPC systems. One approach is based on partitioning the networks and providing an isolated partition for a job. While this approach has successfully worked for low-radix networks such as torus [87], it is a challenge to implement partitioning on networks such as dragonfly or megafly due to their hierarchical nature. The other approach is QoS, which can be enforced through various mechanisms on data centers and HPC networks. Flow control [88], [89] is a high-level approach for avoiding interference in large-scale and data center-scale networks which takes a coarser-grained look at data within the network. Alizedah et al. [90] studied the impacts of sacrificing a portion of the total bandwidth while lowering the threshold for congestion sensing to provide a buffer zone within links in an attempt to reduce the overall latency of applications in a data center environment. On the algorithmic routing side of QoS implementation, many different approaches exist, from centralized global information methods to distributed routing algorithms with limited or incomplete network information and hierarchal algorithms that bridge the gap between globally and locally available information when making routing decisions. Chen and Nahrstedt [18] presented an overview of various routing algorithms solving different QoS problems for both unicast and multicast applications. Most of the literature available on quality of service is intended for data-centric and TCP/IP networks

Portions of this chapter previously appeared as: M. Mubarak, N. McGlohon, *et al.* "Evaluating quality of service traffic classes on the megafly network," in *Proc. Int. Conf. High Perform. Comput.*, 2019, pp. 3–20.

Portions of this chapter previously appeared as: N. McGlohon, N. Wolfe, M. Mubarak, and C. D. Carothers, "Fit Fly: a case study on interconnect innovation through parallel simulation," in *Proc. ACM SIGSIM. Conf. Princ. Adv. Discrete Simul.*, 2019, pp. 137–148.

Portions of this chapter previously appeared as: N. McGlohon, R. B. Ross, and C. D. Carothers, "Evaluation of link failure resilience in multirail dragonfly-class networks through simulation," in *Proc. ACM SIGSIM. Conf. Princ. Adv. Discrete Simul.*, 2020, pp. 105–116.

Portions of this chapter have been submitted to: N. McGlohon *et al.*, "Exploration of congestion control techniques on dragonfly-class HPC networks through simulation," in *2021 IEEE Int. Conf. Cluster Comput.*

Portions of this chapter previously appeared as: N. McGlohon and C. D. Carothers, "Unbiased deterministic total ordering of parallel simulations with simultaneous events,", May 2021, arXiv:2105.00069v1 [cs.DC].

Portions of this chapter are to appear in: N. McGlohon and C. D. Carothers, "Toward unbiased deterministic total ordering of parallel simulations with simultaneous events," in *Proc. 2021 Winter Simul. Conf.*

and does not explore HPC workloads, routing, and flow control mechanisms. Cheng et al. [41] provided high-level details about implementing quality of service on data-centric and HPC networks. The work is not specific to topology and does not provide any performance results. Jakanovic et al. [91] provided an efficient QoS policy for HPC systems with InifiBand network (fat-tree topology).

9.2 On Innovation Through Simulation and Multi-Rail Networks

Work on simulating dual-rail dual-plane fat-tree networks [51] identified many potential benefits to the extra bandwidth and router-buffer space supplied by a second, independent, plane of routers. The authors did mention, however, that whether any performance benefits were realized would depend on the supplied workload.

Fit Fly is a variant of the Slim Fly topology [16], which is a network class that features a low diameter. In their work, the authors show that lowering the diameter of a network can reduce not only the latency of operations on the network but also the cost of the network in terms of both construction and operating energy.

Being able to answer hypothetical "what-if?" questions was a motivating point for the CODES toolkit [62], and evidence supporting the value of these types of ventures has been presented in several works. In [21], the authors describe tests of various networks on varying workloads, including synthetic and HPC network traces. They show that CODES can be used by network designers to gain crucial insight into prospective network designs in an efficient manner.

The authors of [45] argue that testing networks with a diverse set of workloads and configurations is important to obtaining reliable predictions of real-world network performance. They introduce TraceR and use it to add support for more types of HPC traces into CODES.

In addition to the CODES simulator used in this work, there are many other network simulation tools with similar goals and uses.

BigNetSim [92], a PDES system based on the Pose simulation environment, provides packet-level simulation with sequential, conservative parallel, and optimistic parallel execution modes. It features several synthetic traffic generators as well as application trace support.

INSEE [93] is a network simulation and evaluation environment that joins a functional simulator and traffic generator into one simulation system. While it only operates under sequential execution, it consumes very little memory due to its functional nature. The authors

note that a 64,000 node simulation was run while consuming only 2GB of memory.

BookSim [94] is a sequential-execution-only network simulator with similar traffic generation and application trace features. While it only works sequentially, it is a *cycle-accurate* flit-level simulator making its results very reliable. This does, however, limit the overall size of simulatable networks.

More information on various tools and systems for simulating interconnection networks, as well as associated challenges, can be found in [95].

9.3 On Link Failure Effects and Mitigation

Similar work observing the effects of link failure has been performed on single-planar 1D-Dragonfly networks [96]. The authors of that work tested an 8,192 node network with 1% link failure rate with three workloads using the Structural Simulation Toolkit network simulator [97]. They encountered similar issues in the realm of efficient routing computations; they precomputed a small subset of paths from a network with no link failures and filtered out paths that had failed links. A limitation of their work is that a legal path may have existed between two nodes in the network but was not included in the subset: thus, if all paths in the subset were rendered invalid by link failure, communication would fail. In contrast, our work dynamically determines legal paths to route along if they exist.

In recent years, studies have sought topologies that are particularly resilient to router link failures. Modified versions of the fat-tree network topology have been tested with varying levels of link failure and showed improvement over the default fat-tree network [98], [99].

Routing correctly in an HPC network is crucial to avoiding deadlock and performance-limiting situations, and this is even more important in irregular networks such as those with link failures. Many studies [98]–[103] have been performed on routing deadlock-free within a network with link failures.

Another way to route correctly within a network with failed links is to completely ignore the type of network it originally was, treating the new, irregular network as an arbitrary graph. This type of routing is known as topology-agnostic routing. A survey of routing algorithms for general networks discusses the challenges of routing in irregular interconnection networks without deadlock [104]. Often algorithms of this type are also called oblivious because the routing decisions of one packet do not affect the routing decisions of another. Work has shown that the problem of finding an appropriate set of virtual channels in order

to route, deadlock-free, for an entirely arbitrary graph is NP-complete [105], and heuristics have been provided to solve the problem.

The CODES simulator has a large body of work associated with it. Numerous researchers have utilized the CODES simulator for job-interference and performance studies with HPC application traces. Recent works following that trend include adding and evaluating new CODES network models [40], [48], [50], [106], expanding CODES to allow for quality-of-service traffic classes in Dragonfly models [40], and extending CODES network models to allow for multirail-multiplanar configurations of fat-tree and Slim Fly [51], [63], [107]. Similar to the methodology of this work, each of these other CODES-based works analyzes the performance of different interconnect network models with various benchmark workloads to exhibit the features of their specific contributions.

Other recent works with simulating the HPC application communication include an analysis of different network topologies with three dimensions of variability, also performed with the CODES simulator [108]. While the CODES simulator focuses on replaying traces of HPC applications, other tools, such as PyPassT [109], analyze parallel application source code and transform it into workloads for their Performance Prediction Toolkit to generate realistic performance estimations.

9.4 On Congestion Control

The topic of congestion control has existed for some time. The problem of congestion in the modern understanding of network switches has existed since the dawn of IP/TCP protocols [110]. Numerous solutions have been proposed and surveyed for general switching networks [111]. TCP, the protocol acting as the backbone for most data transmitted between nodes in the public internet, has historically been the main focus for the development of congestion control mechanisms. There are many actions employed by TCP/IP networks to prevent and mitigate congestion. Surveys of these techniques can be found in [112], [113]. Common actions by which TCP networks commonly recognize and address congestion are through Explicit Congestion Notification (ECN) [114] and dropping packets.

HPC networks, however, operate in a different space. Some difficulties of TCP/IP networking, such as relying on transmission of packets to and from devices in vastly different locales and through switches owned and operated by neither the source nor destination, have no analogs in high-performance interconnects. Many HPC interconnect fabrics, like

Mellanox Infiniband [57], try to avoid the dropping of packets, and when they do it is usually as a last resort to resolve an error. So while in TCP networks, congestion can be, however aggressively, removed by simply dropping all problem packets. This is not preferred in HPC interconnects and is generally avoided. There are mechanisms proposed, however, such as Speculative Reservation Protocol (SRP), Small-Message SRP (SMSRP), and Last-Hop Reservation Protocol (LHRP) which do rely on the ability to drop problematic messages [115], [116] that could be employed in HPC interconnects.

Infiniband, however, is considered a loss-less interconnect and must assure that packets not be dropped except in the case of component failure, and their Infiniband Congestion Control Architecture (CCA) [117] relies on the communication of ECN messages to and from endpoints to throttle their injection and slowly ease up over time. The effects and tuning of the CCA system have been explored by numerous works [118]–[121].

The authors of [122] propose an interesting solution by introducing flow isolation in addition to injection throttling, bringing in QoS-like techniques for reducing the impact and severity of congestion.

The approach to congestion control taken in Chapter 5 was inspired by publicly available patents from HPE Cray [67]–[69]. Guidance for the creation of the congestion control approach in Chapter 6 was found in a 2020 keynote [123] and in the short description of the Slingshot interconnect congestion control system [72].

9.5 On General Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) systems and the models developed for them have been used to explore a wide variety of topics. The PDES engine that is utilized in this work, Rensselaer's Optimistic Simulation System (ROSS) [24], is the focus of significant research itself.

Parallel simulation offers significant improvements over standard sequential simulation as it can distribute computational load across a massive compute node pool. This distribution comes with some overhead. With simulation processes spread out multiple processors, natural forces like clock drift can cause issues and lead to out-of-order events that can render a simulation invalid if not avoided or properly handled. Conservative parallel execution, a type of parallel simulation which utilizes synchronization overhead up front to make it impossible for any processes to schedule events that would be received/processed out of order. Algorithms

for this type of execution include, but are not limited to, the Null Message algorithm [26], [124], the YAWNS protocol [125]–[127], and Composite Synchronization [128].

Another school of parallel simulation is optimistic execution, which allows for processes to process events as they receive them and at their own pace. Instead of applying overhead upfront to prevent out-of-order events from occurring, it incurs overhead in the event of out-of-order events being detected and resolved. For ROSS' optimistic execution mode, it implements reverse computation [27] to allow for simulation entities to roll back to previous, known valid states in the event of discovered out-of-order events resulting from optimistic parallel execution. This method has been shown in several counts as being highly scalable, allowing for massively parallel simulations to be executed [21], [50], [60], [83].

Optimistic execution generally yields considerable improvement over conservative parallel execution. Optimistic execution's superior performance relies on the assumption that out-of-order events are relatively uncommon. If out-of-order events become too common, then a considerable amount of computation time will be spent rolling back events just to replay them forward again, constantly backtracking and inhibiting simulation progress. Simulation tuning for both conservative and optimistic execution modes can improve their performance, reducing the impact that the validity overhead has on the runtime performance [129].

Further background on PDES, including associated challenges and details of the topic domain can be found in Richard Fujimoto's survey on the subject [25].

9.6 On Virtual Time and Event Simultaneity

The topic of event simultaneity and ordering in parallel simulations has been discussed in various contexts over the span of decades [80], [86], [130]–[135]. In [80], the topic of distributed logical and physical clocks, causal ordering conditions and happens-before relations is discussed but notes that the included theory makes no requirement about the ordering of incomparable or simultaneous events. The clocks discussed base their perception of time based on the perceived happens-before relations to other events.

Virtual time [28] takes a slightly different approach by inferring happens-before relations from the encoded timestamps giving a simulated model the power to define when events happen in relation to each other. In [136], the authors also extend ROSS' definition of virtual time to encode lower order bits to break event ties.

Much of this paper was inspired by the work performed in [86]. The author makes

arguments that given a simulation with n simultaneous events, then the *correct* final result is the mean of all $n!$ possible event orderings. Without some mechanism to effectively explore various possible orderings, a correct final result is not achievable. In our work, we provide a method that allows for random sampling of the possible event orderings.

That same year, the authors of [132] proposed an event ordering mechanism for simultaneous events, which also employed an extension to the basic timestamp for encoding event priority. This mechanism provided identical results between a sequential and parallel executed simulation and is similar in practice to our definition of biased random causal ordering.

The authors of [131] establish methods for breaking ties based on a logical clock interpretation of event causality or user-defined priorities. They also briefly argue how different permutations of otherwise incomparable events should be considered and will inherently affect the outcomes of future events but do not elaborate on a solution to effectively explore this space. Similarly, in [133]–[135], the authors note the importance of discovering other potential outcomes of simultaneous event orderings and propose a PDES branching mechanism tool that facilitates this.

CHAPTER 10
CONCLUDING REMARKS

10.1 Broad Discussion

In this document, we've discussed works in three areas focusing on how to effectively understand the effects of congestion in high-performance computing communication interconnect networks, methods of avoiding and resolving it, through parallel discrete event simulation. The work and techniques utilized represent a stepping point upon which greater innovation in the areas of network architecture design and simulation can be facilitated.

10.1.1 Congestion Avoidance

The first line of defense against the effects of HPC communication network congestion is avoiding its occurrence in the first place. While pockets of localized network congestion may be avoided via clever packet routing schemes, they can only do so much good before secondary routes also become congested. In effect, one could consider adaptive routing as a load balancing tool rather than the main mechanism for preventing congestion. Instead, more targeted approaches, starting early on in the design phase of an HPC network interconnect architecture.

The inclusion of Quality of Service (QoS) capabilities via the partitioning of network links to enable queues of different priority levels for different classes of traffic is a common approach to reducing the effects of congested networks. This technique works by giving applications the appearance of operating on their own independent network within their traffic class and imposing limits on bandwidth usage to each class. The result is a more well-behaved network with reduced inter-job packet interference. We've demonstrated two approaches to QoS in HPC networks, one by assigning explicit priority levels to individual applications operating on the network and another by assigning higher priority to collective communication operations requiring participation from many processes across the network. The first can be more effective but yields the problem of determining who gets – and how much – priority bandwidth access. The second can take a more application-agnostic approach and yield less dramatic results but can be broadly applied to sets of applications with unbounded cardinality.

While QoS has its benefits, it can be difficult to *effectively* configure for networks with

hundreds of different applications all competing for the same resources. This contention for resources is often focused on one particular location: the memory buffers containing the queue of packets waiting to be routed onward from a network switch. Packet interference occurs when two (or more) packets find themselves waiting in the same queue. The latency of the communication that they are facilitating will depend on how quickly they reach the front of the queue and is then correlated with who is forwarded onward first.

Thus, reducing the probability that any two packets find themselves in the same memory buffers will likely reduce the occurrence of congestion and its associated packet interference. One solution would just be to design a larger network, but there is an upper limit to network size due to the maximum radix, or number of ports, available on current generation switches. To achieve a 'bigger' network without the need for higher radix switches, multi-planar networks, those with multiple independent planes/sub-networks of switches, can be designed. These networks give endpoints several choices where to inject traffic and multiply the number of paths available to a packet.

Congestion can manifest from a variety of sources. Most commonly, it occurs when applications submit more traffic than the network can effectively handle, overwhelming the queues within network switches with the end result of higher communication latencies. Even a network designed with adequate resource provisioning can fall victim to network congestion when inevitable network component failure eventually occurs. When the links connecting switches to each other fail, the number of valid paths connecting two endpoints decreases, and the network starts routing the same number of packets through a smaller set of paths. Reducing the impact of this occurrence buys system administrators more time – allowing them to resolve the issue during regularly scheduled system downtime. Thus, designing a network architecture that is *resilient* to link failure is important to avoiding the impact from them when – not if – they occur. Networks with high path diversity, including multi-planar networks, are more likely to not suffer significantly when a link fails.

10.1.2 Congestion Control

Once congestion manifests in a communication network, it can be difficult to treat. Referring back to the analogy of vehicular traffic in road networks, as long as more cars continue to embark on new routes, then congestion will likely not resolve on its own. Once the period of high traffic input – rush hour, for example – is over, congestion will eventually

dissipate naturally. The concept of congestion control in high-performance computing communication interconnect networks operates on the same principle: reducing the number of packets being injected into the network during times of recognized congestion.

There is an immediate problem with this: unnecessarily applying a throttling of packet injection to well-behaved applications or ranks within them can – and most likely will – affect their overall performance. This can lead to longer application runtimes, increased operation latencies, and lower overall system utilization. Thus, the key to effective congestion control is applying targeted throttling and abatement techniques with a strength that is proportional to the individual process's contribution to the identified congestion.

We've explored two mechanisms for detecting, identifying the cause of, and treating congestion in HPC communication networks. One, based on public patents on the topic applied to current (at the time) generation interconnect architectures, attempted to detect whether the network was in a state of broad congestion, identifying the primary causal applications, and throttling them accordingly. This strategy employed a management layer with global information collected from switches and terminals in the network. Unfortunately, this strategy was very complex to tune and was slow to recognize congestion and too heavy-handed in its approach to treating it and led to less than optimal results.

The second was a more distributed approach, giving every switch in the network the authority to determine when it was experiencing congestion, identifying the exact set of processes with packets on its buffers contributing to the congestion, and sending signals to those particular application ranks to throttle. By enabling the ability for the network endpoints facilitating the injection/ejection of packets to/from the network the knowledge of how quickly their previously injected packets are ejected from the network, these terminals can self limit their injection to be no greater than the rate at which their packets are ejected from the network. This naturally allows the network to only accept exactly as much traffic as it can effectively handle, resulting in a stable level of controlled network utilization.

10.1.3 Effective Simulation

While the determinism that is featured by ROSS and many parallel discrete event simulation frameworks and engines is beneficial to provide, when allowable by the simulated model's requirements, it comes with a cost. Every single result achieved from its execution will be exactly the same. In order to combat this, it is common for simulation engines to

allow for the variance of the pseudo-random generator seed that produces any model-level randomness.

Changing the seed will likely result in a different final simulation outcome, and we can run many different seeds to sample from the space of all simulated model possibilities but with one major limitation. Unless a fair, unbiased, way of determining the order of simultaneous (in virtual time) events is established by the engine, any statistical aggregation of multiple differently seeded simulations will be inherently biased and thus flawed.

We've provided three mechanisms for breaking ties: one biased and two unbiased, while also accounting for the possibility of zero-offset events scheduled to occur at the same time as its creator. We have analyzed and exhibited the performance of this mechanism in numerous ROSS simulations and argued for the necessity should any statistical analysis of multiple simulations be desired.

Furthermore, we've demonstrated the effectiveness of this new feature of ROSS in a series of CODES HPC interconnect network simulations. This is a substantial step forward for the simulation of network interconnects for a number of reasons. Firstly, it makes it easier to develop simulation models by reducing the likelihood of observing non-determinism which had spawned from non-deterministic ordering of simultaneous events. Otherwise, this might be mistakenly used to justify the belief that a bug in the system exists. Secondly, it allows for a simulated model to decouple event timing, which is commonly used as a metric for model measurements, from any event-tie avoidance techniques. And lastly, we've shown that it can be used to sample from the space of all possible simulation event occurrences allowable by the simulated model allowing for a deeper understanding of general behavior.

All of this together allows for a higher degree of confidence in the simulated model and can easily verify how representative any observed measurements are of the expectation.

10.2 Future Work

The vast amount of experiments presented in this work reflects only a fraction of the possible routes of exploration given the developments created during the course of these studies. Different workloads, communication patterns, routing algorithms, quality-of-service configurations, congestion control thresholds and parameters, can each be explored in tandem with one another.

Furthermore, the work presented in Part III could be applied to the above variations

and yield even *more* useful data with which to gain deeper insight into the behaviors and dynamics of these systems.

That being said, there are some prime candidates for future exploration that are of particular interest.

10.2.1 Quality of Service

There have been recent developments in the CODES implementation of Quality of Service that makes it much more realistic, capable, and flexible than what was presented in Chapter 2. This has included the use of a more dynamic token-bucket rate-limiting mechanism that creates a more reliable mechanism for limitation than monitoring bandwidth over static windows.

The developer who made the above change also created a more adaptive configuration interface for QoS, which makes it easier to assign traffic classes to large numbers of jobs. The exploration of QoS in this work was limited to two traffic classes with a maximum of three simultaneous workloads.

10.2.2 Topology Agnostic Routing

During the work exhibited in Chapter 4, there was a brief exploration into the topic of topology agnostic routing. This is an interesting concept as it could pave the way for a generalized network simulation model. Current CODES networks are wholely described within their own source files, which has led to a lot of replicated code and is an environment that is prone to semantic coding errors.

This problem becomes worse for every single adaptation of a network model to another. Topology agnostic routing algorithms would allow for prototypical CODES models to be represented as files containing the interconnection links. This is unlikely to be as *accurate* as a dedicated model but could be utilized to gain faster insight into the capabilities of a given topology over another.

10.2.3 Congestion Control

The topic of congestion control was explored with significant depth in Part II, but it remains a prime contender for managing and preventing congestion in current and future state-of-the-art interconnection networks. This work focused on the many-to-one incast traffic

pattern, but it is unlikely to be the only problematic pattern in existence today. Further research into problematic traffic patterns could be explored with CODES and its SWM workload generation tools.

Despite the depth of Chapter 6, there are a few parameters for the configuration of ***CODES-CC2*** that could prove beneficial to optimize. Full experimental sweeps of these parameters, including congestion/decongestion thresholds and the measurement period, could yield significant improvements to the results presented here.

QoS and congestion control tackle a similar problem – or at least the effects of their mechanisms on application performance are similar. However, their approaches to the reduction of congestion in a network are entirely different. Furthermore, both of these technologies are likely to co-exist in a real-world HPC system. We would like to explore the interaction of these two mechanisms and the dynamics that result. Could low-priority traffic classes be a helpful place to "quarantine" identified problematic workloads? How well does QoS handle many-to-one incast traffic patterns?

10.2.4 Impact of Simultaneous Events

In Part III, we explored the concept of simultaneous events and how to handle them fairly and deterministically. We also looked into the implications that this work has on the simulation of HPC networks. This opens a significant amount of research opportunities for interconnection simulation.

The capability of exploring other potential orderings of simultaneous events allows for significant amounts of statistical data about the network model to be gathered. This data could be useful for the training of a low-fidelity network simulation tool that can capture the broad-scale behavior of a network model and extend it to much higher-magnitude time scales. This enables a completely different approach to how network simulation can be utilized to generate performance expectations of future massive-scale systems.

10.3 Conclusion

In summary, what we've shown in this body of work is the impact that congestion can have on a network once it manifests. We've shown methods for designing networks to avoid it or reduce its likelihood of occurrence. Similarly, we've demonstrated two similar mechanisms for controlling and managing congestion once it is discovered and showcased their capabilities.

Realizing the first exascale-power supercomputer will require some degree of mastery over congestion and significant innovation in the design of the communication network interconnect backbone. Simulation is an important tool in predicting the behavior of proposed designs, gaining accurate metrics for informed guidance in the process.

Additionally, simulation itself as a topic continues an area of fruitful research and innovation. New techniques grant us a deeper level of understanding of simulated models and a higher level of confidence in their reported results.

With scalable, effective simulation running on ever-more-powerful supercomputers, the harder problems inside and outside of computing will become that much easier to solve.

REFERENCES

[1] U. S. Census Bureau, "Average travel time to work in the united states by metro area," census.gov, https://www.census.gov/library/visualizations/interactive/work-travel-time.html (accessed Nov 7, 2020).

[2] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Trans. Comput.*, vol. C-34, no. 10, pp. 892–901, Oct. 1985.

[3] J. Wells, B. Bland, J. Nichols, J. Hack, F. Foertter, G. Hagen, T. Maier, M. Ashfaq, B. Messer, and S. Parete-Koon, "Announcing supercomputer summit," Oak Ridge Nat. Lab., Oak Ridge, TN, USA, Tech. Rep., 2016.

[4] C. B. Stunkel, R. L. Graham, G. Shainer, M. Kagan, S. S. Sharkawi, B. Rosenburg, and G. A. Chochia, "The high-speed networks of the summit and sierra supercomputers," *IBM J. Res. Dev.*, vol. 64, no. 3/4, pp. 3:1–3:10, Jan. 2020.

[5] Center for Computational Innovations, "Artificial intelligence multiprocessing optimized system (AiMOS)," cci.rpi.edu, https://cci.rpi.edu/aimos (accessed Nov 1, 2020).

[6] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, and T. Inoue, "The Tofu Interconnect D," in *IEEE Int. Conf. Cluster Comput.*, 2018, pp. 646–654.

[7] R. E. Kessler and J. L. Schwarzmeier, "Cray T3D: a new dimension for cray research," in *Dig. of Papers. COMPCON Spring*, 1993, pp. 176–182.

[8] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken *et al.*, "Blue Gene/L torus interconnection network," *IBM J. Res. Dev.*, vol. 49, no. 2.3, pp. 265–276, Mar. 2005.

[9] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay *et al.*, "Overview of the Blue Gene/L system architecture," *IBM J. Res. Dev.*, vol. 49, no. 2.3, pp. 195–212, 2005.

[10] IBM, "Overview of the ibm Blue Gene/P project," *IBM J. Res. Dev.*, vol. 52, no. 1.2, pp. 199–220, Jan. 2008.

[11] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow *et al.*, "Looking under the hood of the IBM Blue Gene/Q network," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2012, pp. 1–12.

[12] 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 *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2011, pp. 1–10.

[13] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *IEEE Int. Symp. Comput. Architecture*, 2008, pp. 77–88.

[14] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: a scalable HPC system based on a dragonfly network," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2012, pp. 1–9.

[15] A. Shpiner, Z. Haramaty, S. Eliad, V. Zdornov, B. Gafni, and E. Zahavi, "Dragonfly+: low cost topology for scaling datacenters," in *IEEE Workshop High-Perform. Interconnection Netw. Exascale Big-Data Era*, 2017, pp. 1–8.

[16] M. Besta and T. Hoefler, "Slim fly: a cost effective low-diameter network topology," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2014, pp. 348–359.

[17] L. G. Valiant, "A scheme for fast parallel communication," *SIAM J. Comput.*, vol. 11, no. 2, pp. 350–361, Oct. 1982.

[18] S. Chen and K. Nahrstedt, "An overview of quality of service routing for next-generation high-speed networks: problems and solutions," *IEEE Network*, vol. 12, no. 6, pp. 64–79, Nov. 1998.

[19] M. Mubarak, P. Carns, J. Jenkins, J. K. Li, N. Jain, S. Snyder, R. Ross, C. D. Carothers, A. Bhatele, and K.-L. Ma, "Quantifying I/O and communication traffic interference on dragonfly networks equipped with burst buffers," in *IEEE Int. Conf. Cluster Comput.*, 2017, pp. 204–215.

[20] S. Snyder, P. Carns, J. Jenkins, K. Harms, R. Ross, M. Mubarak, and C. Carothers, "A case for epidemic fault detection and group membership in HPC storage systems," in *Int. Workshop on Perform. Model., Benchmarking and Simul. of High Perform. Comput. Syst.*, 2014, pp. 237–248.

[21] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "Enabling parallel simulation of large-scale HPC network systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 87–100, Jan. 2017.

[22] M. Mubarak and R. B. Ross, "Validation study of codes dragonfly network model with theta Cray XC system," Argonne Nat. Lab., Argonne, IL, USA, Tech. Rep. ANL/MCS-TM-369135666, 2017.

[23] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *IEEE Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–11.

[24] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: a high-performance, low-memory, modular time warp system," *J Parallel Distr. Comput.*, vol. 62, no. 11, pp. 1648–1669, Nov. 2002.

[25] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[26] K. M. Chandy and J. Misra, "Distributed simulation: a case study in design and verification of distributed programs," *IEEE Trans. Softw. Eng.*, no. 5, pp. 440–452, Sep. 1979.

[27] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, "Efficient optimistic parallel simulations using reverse computation," *ACM Trans. Model. Comput. Simul.*, vol. 9, no. 3, pp. 224–253, Jul. 1999.

[28] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985.

[29] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Elsevier, 2004.

[30] Sandia National Laboratories, "SST-DUMPI," GitHub.com, https://github.com/sstsimulator/sst-dumpi (accessed Dec. 10, 2018).

[31] U.S. Dept. of Energy, "Characterization of the DOE mini-apps," NERSC.gov, https://portal.nersc.gov/project/CAL/doe-miniapps.htm (accessed Jul 14, 2016).

[32] J. Thompson, "Scalable workload models for system simulations," [Powerpoint Slides], 2014, https://hpc.pnl.gov/modsim/2014/Presentations/Thompson.pdf (accessed: Nov. 1, 2018).

[33] National Energy Research Scientific Computing Center, "Cori," NERSC.gov, https://docs.nersc.gov/systems/cori/ (accessed May 5, 2021).

[34] Los Alamos National Laboratory, "Trinity Cray XC40 system," lanl.gov, http://www.lanl.gov/projects/trinity/ (accessed May 5, 2021).

[35] Argonne Leadership Computing Facility, "Theta, argonne's Cray XC system," alcf.anl.gov, https://www.alcf.anl.gov/theta (accessed Mar. 1, 2019).

[36] S. Chunduri, K. Harms, S. Parker, V. Morozov, S. Oshin, N. Cherukuri, and K. Kumaran, "Run-to-run variability on xeon phi based Cray XC systems," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2017, pp. 1–13.

[37] M. Flajslik, E. Borch, and M. A. Parker, "Megafly: a topology for exascale systems," in *Int. Conf. High Perform. Comput.*, 2018, pp. 289–310.

[38] X. Yang, J. Jenkins, M. Mubarak, R. B. Ross, and Z. Lan, "Watch out for the bully! job interference study on dragonfly network," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2016, pp. 750–760.

[39] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of mpi usage on a production supercomputer," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2018, pp. 386–400.

[40] M. Mubarak, N. McGlohon, M. Musleh, E. Borch, R. B. Ross, R. Huggahalli, S. Chunduri, S. Parker, C. D. Carothers, and K. Kumaran, "Evaluating quality of service traffic classes on the Megafly network," in *Int. Conf. High Perform. Comput.*, 2019, pp. 3–20.

[41] A. S. Cheng, T. D. Lovett, and M. A. Parker, "Traffic class arbitration based on priority and bandwidth allocation," U.S. Patent 10,237,191, Mar., 2019.

[42] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault *et al.*, "Argobots: a lightweight low-level threading and tasking framework," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 512–526, Mar. 2018.

[43] M. Mubarak, N. McGlohon, M. Musleh, E. Borch, R. Ross, R. Huggahalli, S. Chunduri, S. Parker, C. Carothers, and K. Kalyan, "Exascale computing project hardware evaluation milestone report: evaluating quality of service on high-radix HPC networks," U.S. Dept. of Energy, Washington D.C., USA, Tech. Rep., 2018.

[44] J. Won, G. Kim, J. Kim, T. Jiang, M. Parker, and S. Scott, "Overcoming far-end congestion in large-scale networks," in *IEEE Int. Symp. High Perform. Comput. Architecture*, 2015, pp. 415–427.

[45] N. Jain, A. Bhatele, S. T. White, T. Gamblin, and L. V. Kale, "Evaluating HPC networks via simulation of parallel workloads," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2016, pp. 154–165.

[46] M. Garcia, E. Vallejo, R. Beivide, M. Odriozola, C. Camarero, M. Valero, J. Labarta, C. Minkenberg *et al.*, "On-the-fly adaptive routing in high-radix hierarchical networks," in *IEEE Int. Conf. Parallel Process.*, 2012, pp. 279–288.

[47] A. Singh, "Load-balanced routing in interconnection networks," Ph.D. dissertation, Dept. Elect. Eng. Stanford University, Stanford, CA, 2005.

[48] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "Modeling a million-node dragonfly network using massively parallel discrete-event simulation," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2012, pp. 366–376.

[49] ——, "Using massively parallel simulation for MPI collective communication modeling in extreme-scale networks," in *Proc. of the 2014 Winter Simulation Conf.*, pp. 3107–3118.

[50] N. Wolfe, C. D. Carothers, M. Mubarak, R. Ross, and P. Carns, "Modeling a million-node slim fly network using parallel discrete-event simulation," in *ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2016, pp. 189–199.

[51] N. Jain, A. Bhatele, L. H. Howell, D. Böhme, I. Karlin, E. A. León, M. Mubarak, N. Wolfe, T. Gamblin, and M. L. Leininger, "Predicting the performance impact of different fat-tree configurations," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2017, pp. 1–13.

[52] M. Miller and J. Sirán, "Moore graphs and beyond: a survey of the degree/diameter problem," *Elect. J. Combinatorics*, no. DS14, p. 92, May 2012.

[53] B. D. McKay, M. Miller, and J. Širáň, "A note on large graphs of diameter two and given maximum degree," *J. Combinatorial Theory, Series B*, vol. 74, no. 1, pp. 110–118, Sep. 1998.

[54] G. Kathareios, C. Minkenberg, B. Prisacari, G. Rodriguez, and T. Hoefler, "Cost-effective diameter-two topologies: analysis and evaluation," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2015, pp. 1–11.

[55] Lawrence Berkeley National Laboratory, "Boxlib codes," GitHub.com, https://boxlib-codes.github.io/ (accessed Mar. 14, 2019).

[56] HPE Cray Inc., "Meet slingshot: an innovative interconnect for the next generation of supercomputers," Cray.com, https://www.cray.com/blog/meet-slingshot-an-innovative-interconnect-for-the-next-generation-of-supercomputers (accessed Oct. 30, 2018).

[57] Mellanox Technologies, "The official store," Mellanox.com, https://store.mellanox.com/ (accessed Mar. 14, 2019).

[58] C. J. Ross, C. D. Carothers, M. Mubarak, R. B. Ross, J. K. Li, and K.-L. Ma, "Leveraging shared memory in the ross time warp simulator for complex network simulations," in *IEEE 2018 Winter Simul. Conf.*, pp. 3837–3848.

[59] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "A case study in using massively parallel simulation for extreme-scale torus network codesign," in *ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2014, pp. 27–38.

[60] N. Wolfe, M. Mubarak, C. D. Carothers, R. B. Ross, and P. H. Carns, "Modeling large-scale slim fly networks using parallel discrete-event simulation," *ACM Trans. Model. Comput. Simul.*, vol. 28, no. 4, pp. 1–25, Aug. 2018.

[61] W. J. Dally, "Virtual-channel flow control," *ACM SIGARCH Comput. Architecture News*, vol. 18, no. 2SI, pp. 60–68, May 1990.

[62] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross, "CODES: enabling co-design of multilayer exascale storage architectures," in *Proc. Workshop Emerging Supercomputing Tech.*, 2011, p. 6.

[63] N. McGlohon, N. Wolfe, M. Mubarak, and C. D. Carothers, "Fit fly: a case study on interconnect innovation through parallel simulation," in *ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2019, pp. 137–148.

[64] G. Maglione-Mathey, P. Yebenes, J. Escudero-Sahuquillo, P. J. Garcia, F. J. Quiles, and E. Zahavi, "Scalable deadlock-free deterministic minimal-path routing engine for infiniband-based dragonfly networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 183–197, Jan. 2018.

[65] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms, Second Edition*. Cambridge, MA, USA: MIT Press, 2001.

[66] N. Jiang, J. Kim, and W. J. Dally, "Indirect adaptive routing on large scale interconnection networks," in *Int. Symp. Comput. Architecture*, 2009, pp. 220–231.

[67] L. S. Kaplan, E. L. Froese, C. B. Johns, M. P. Kelly, A. F. Godfrey, and B. T. Shields, "Congestion detection in a network interconnect," U.S. Patent 9 391 899 B2, Jul., 2016.

[68] ——, "Congestion causation in a network interconnect," U.S. Patent 9 674 091 B2, Jun., 2017.

[69] E. L. Froese, C. B. Johns, A. F. Godfrey, L. S. Kaplan, M. P. Kelly, and B. T. Shields, "Congestion abatement in a network interconnect," U.S. Patent 9 674 092 B2, Jun., 2017.

[70] U. S. Dept. of Energy, "Exascale computing project," exascaleproject.org, https://www.exascaleproject.org/ (accessed: Jan 3, 2021).

[71] HPE Cray Inc., "HPE slingshot interconnect," hpe.com, https://www.hpe.com/us/en/compute/hpc/slingshot-interconnect.html (accessed Mar. 1, 2020).

[72] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An in-depth analysis of the slingshot interconnect," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2020, pp. 1–14.

[73] G. Kim, C. Kim, J. Jeong, M. Parker, and J. Kim, "Contention-based congestion management in large-scale networks," in *IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–13.

[74] L. De Haan, A. Ferreira, and A. Ferreira, *Extreme Value Theory: An Introduction*. Cham, Switzerland: Springer, 2006, vol. 21.

[75] P. J. Rousseeuw and M. Hubert, "Robust statistics for outlier detection," *Wiley Interdiscip. Rev.: Data Min. Knowl. Discov.*, vol. 1, no. 1, pp. 73–79, Jan. 2011.

[76] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: a high-performance, low-memory, modular time warp system," in *Proc. Workshop on Parallel and Distrib. Simulation*, 2000, pp. 53–60.

[77] D. Jefferson and H. Sowizral, "Fast concurrent simulation using the time warp mechanism," Rand. Corp., Santa Monica, CA, USA, Tech. Rep. ADA129431, 1985.

[78] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. D. Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, "Distributed simulation and the time warp operating system," NASA Jet Propulsion Laboratory, Pasadena, CA, USA, Tech. Rep. OSTI 5639121, 1987.

[79] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. D. Loreto, "Time warp operating system," *ACM Symp. Operating Syst. Rev.*, vol. 21, no. 5, pp. 77–93, Nov. 1987.

[80] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 179–196, Oct. 1978.

[81] N. L. Johnson, S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions, Volume 2, Second Edition*. Hoboken, NJ, USA: John Wiley & Sons, 1995.

[82] R. M. Fujimoto, "Performance of time warp under synthetic workloads," in *Proc. SCS Multiconference on Dist. Simul.*, 1990, pp. 23–28.

[83] P. D. Barnes Jr, C. D. Carothers, D. R. Jefferson, and J. M. LaPre, "Warp speed: executing time warp on 1,966,080 cores," in *ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2013, pp. 327–336.

[84] E. N. Lorenz, "Deterministic nonperiodic flow," *J. Atmos. Sci.*, vol. 20, no. 2, pp. 130–141, Mar. 1963.

[85] CODES-Org, "CODES: enabling co-design of multilayer exascale storage architectures," Github.com, https://github.com/codes-org/codes (accessed Nov. 1, 2019).

[86] F. Wieland, "The threshold of event simultaneity," in *Proc. 11th Workshop Parallel Dist. Simul.*, 1997, pp. 56–59.

[87] Z. Zhou, X. Yang, Z. Lan, P. Rich, W. Tang, V. Morozov, and N. Desai, "Improving batch scheduling on Blue Gene/Q by relaxing 5d torus network allocation constraints," in *IEEE Int. Parallel Dist. Process. Symp.*, 2015, pp. 439–448.

[88] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Comp. Comm. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[89] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *Proc. ACM SIGCOMM 2011 Conference*, pp. 254–265.

[90] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: trading a little bandwidth for ultra-low latency in the data center," in *9th USENIX Symp. Netw. Syst. Des. Impl.*, 2012, pp. 253–266.

[91] A. Jokanovic, J. C. Sancho, J. Labarta, G. Rodriguez, and C. Minkenberg, "Effective quality-of-service policy for capacity high-performance computing systems," in *IEEE Int. Conf. High Perf. Comput. Comm. and IEEE Int. Conf. Embedded Soft. Syst.*, 2012, pp. 598–607.

[92] N. Choudhury, Y. Mehta, T. L. Wilmarth, E. J. Bohm, and L. V. Kalé, "Scaling an optimistic parallel simulation of large-scale interconnection networks," in *IEEE 2005 Winter Simul. Conf.*, 2005, pp. 591–600.

[93] F. J. R. Perez and J. Miguel-Alonso, "Insee: an interconnection network simulation and evaluation environment," in *European Conf. Parallel Process.*, 2005, pp. 1014–1023.

[94] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *IEEE Int. Symp. Perf. Anal. Syst. Soft.*, 2013, pp. 86–96.

[95] K. Ahmed, J. Liu, A.-H. Badawy, and S. Eidenbenz, "A brief history of HPC simulation and future challenges," in *IEEE 2017 Winter Simul. Conf.*, p. 27.

[96] T. Connors, T. Groves, T. Quan, and S. Hemmert, "Simulation framework for studying optical cable failures in dragonfly topologies," in *IEEE Int. Parallel. Dist. Process. Symp. Workshops*, 2019, pp. 859–864.

[97] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis *et al.*, "The structural simulation toolkit," *ACM SIGMETRICS Perf. Eval. Rev.*, vol. 38, no. 4, pp. 37–42, Mar. 2011.

[98] D. F. Bermúdez Garzón, C. G. Requena, M. E. Gómez, P. López, and J. Duato, "A family of fault-tolerant efficient indirect topologies," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 4, pp. 927–940, Apr. 2016.

[99] M. Adda and A. Peratikou, "Routing and fault tolerance in z-fat tree," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2373–2386, Aug. 2017.

[100] P. T. Gaughan and S. Yalamanchili, "A family of fault-tolerant routing protocols for direct multiprocessor networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 5, pp. 482–497, May 1995.

[101] M. E. Gómez, N. A. Nordbotten, J. Flich, P. López, A. Robles, J. Duato, T. Skeie, and O. Lysne, "A routing methodology for achieving fault tolerance in direct networks," *IEEE Trans. Comput.*, vol. 55, no. 4, pp. 400–415, Apr. 2006.

[102] F. O. Sem-Jacobsen, T. Skeie, O. Lysne, and J. Duato, "Dynamic fault tolerance in fat trees," *IEEE Trans. Comput.*, vol. 60, no. 4, pp. 508–525, Apr. 2011.

[103] D. Xiang, B. Li, and Y. Fu, "Fault-tolerant adaptive routing in dragonfly networks," *IEEE Trans. Depend. Sec. Comput.*, vol. 16, no. 2, pp. 259–271, Mar. 2019.

[104] J. Flich, T. Skeie, A. Mejía, O. Lysne, P. López, A. Robles, J. Duato, M. Koibuchi, T. Rokicki, and J. C. Sancho, "A survey and evaluation of topology-agnostic deterministic routing algorithms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 3, pp. 405–425, Mar. 2012.

[105] J. Domke, T. Hoefler, and W. E. Nagel, "Deadlock-free oblivious routing for arbitrary topologies," in *IEEE Int. Parallel. Dist. Process. Symp*, 2011, pp. 616–627.

[106] Y. Kang, X. Wang, N. McGlohon, M. Mubarak, S. Chunduri, and Z. Lan, "Modeling and analysis of application interference on Dragonfly+," in *ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2019, pp. 161–172.

[107] N. Wolfe, M. Mubarak, N. Jain, J. Domke, A. Bhatele, C. D. Carothers, and R. B. Ross, "Preliminary performance analysis of multi-rail fat-tree networks," in *IEEE/ACM Int. Symp. Cluster, Cloud, Grid Comput.*, 2017, pp. 258–261.

[108] A. Bhatele, N. Jain, M. Mubarak, and T. Gamblin, "Analyzing cost-performance tradeoffs of HPC network designs under different constraints using simulations," in *ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2019, pp. 1–12.

[109] M. A. Obaida, J. Liu, G. Chennupati, N. Santhi, and S. Eidenbenz, "Parallel application performance prediction using analysis based models and HPC simulations," in *ACM SIGSIM Conf. Princ. Adv. Discrete Simul.*, 2018, pp. 49–59.

[110] J. Nagle, "Congestion control in IP/TCP internetworks," *ACM SIGCOMM Comput. Comm. Rev.*, vol. 14, no. 4, pp. 11–17, Oct. 1984.

[111] C.-Q. Yang and A. V. S. Reddy, "A taxonomy for congestion control algorithms in packet switching networks," *IEEE Network*, vol. 9, no. 4, pp. 34–45, Jul. 1995.

[112] J. Widmer, R. Denda, and M. Mauve, "A survey on TCP-friendly congestion control," *IEEE Network*, vol. 15, no. 3, pp. 28–37, May 2001.

[113] S. H. Low, F. Paganini, and J. C. Doyle, "Internet congestion control," *IEEE Control Syst. Mag.*, vol. 22, no. 1, pp. 28–43, Feb. 2002.

[114] S. Floyd, "TCP and explicit congestion notification," *ACM SIGCOMM Comput. Comm. Rev.*, vol. 24, no. 5, pp. 8–23, Oct. 1994.

[115] N. Jiang, D. U. Becker, G. Michelogiannakis, and W. J. Dally, "Network congestion avoidance through speculative reservation," in *IEEE Int. Symp. High-Perform. Comput. Architecture*, 2012, pp. 1–12.

[116] N. Jiang, L. Dennison, and W. J. Dally, "Network endpoint congestion control for fine-grained communication," in *IEEE Int. Conf. High Perform. Comput., Netw., Storage, Anal.*, 2015, pp. 1–12.

[117] D. Crupnicoff, S. Das, and E. Zahavi, "Deploying quality of service and congestion control in infiniband-based data center networks," Mellanox Technologies, White Paper, 2005.

[118] J. R. Santos, Y. Turner, and G. Janakiraman, "End-to-end congestion control for infiniband," in *IEEE Joint Conf. Comm. Soc.*, vol. 2, 2003, pp. 1123–1133.

[119] M. Gusat, D. Craddock, W. Denzel, T. Engbersen, N. Ni, G. Pfister, W. Rooney, and J. Duato, "Congestion control in infiniband networks," in *Symp. High Perform. Interconnects*, 2005, pp. 158–159.

[120] G. Pfister, M. Gusat, W. Denzel, D. Craddock, N. Ni, W. Rooney, T. Engbersen, R. Luijten, R. Krishnamurthy, and J. Duato, "Solving hot spot contention using infiniband architecture congestion control," in *Proc. HP-IPC*, 2005, p. 6.

[121] 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 *IEEE Int. Parallel. Dist. Process. Symp. Workshops*, 2010, pp. 1–12.

[122] J. Escudero-Sahuquillo, E. G. Gran, P. J. Garcia, J. Flich, T. Skeie, O. Lysne, F. J. Quiles, and J. Duato, "Combining congested-flow isolation and injection throttling in HPC interconnection networks," in *Int. Conf. Parallel Process.*, 2011, pp. 662–672.

[123] M. Kagan, "The datacenter is a computer," in *IEEE Symp. High-Perform. Interconnects*, 2020, pp. i–ii, DOI: 10.1109/HOTI51249.2020.00009.

[124] R. E. Bryant, "Simulation of packet communication architecture computer systems," Massachusets Institute of Technology, Cambridge Lab for Computer Science, Cambridge, Massachusetts, USA, Tech. Rep. ADA048290, 1977.

[125] D. M. Nicol, "The cost of conservative synchronization in parallel discrete event simulations," *J. ACM*, vol. 40, no. 2, pp. 304–333, Apr. 1993.

[126] D. M. Nicol, C. C. Michael, and I. Patrick, "Efficient aggregation of multiple PLs in distributed memory parallel simulations," in *IEEE 1989 Winter Simul. Conf.*, pp. 680–685.

[127] P. M. Dickens, D. M. Nicol, P. F. Reynolds Jr, and J. M. Duva, "Analysis of bounded time warp and comparison with YAWNS," *ACM Trans. Model. Comput. Simul.*, vol. 6, no. 4, pp. 297–320, Oct. 1996.

[128] D. M. Nicol and J. Liu, "Composite synchronization in parallel discrete-event simulation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 5, pp. 433–446, May 2002.

[129] C. D. Carothers and K. S. Perumalla, "On deciding between conservative and optimistic approaches on massively parallel platforms," in *IEEE 2010 Winter Simul. Conf.*, pp. 678–687.

[130] H. Rajaei, "Design issues in parallel simulation languages," *IEEE Des. Test. Comput.*, vol. 10, no. 4, pp. 52–63, Dec. 1993.

[131] R. Ronngren and M. Liljenstam, "On event ordering in parallel discrete event simulation," in *Proc. Workshop Parallel Dist. Simul.*, 1999, pp. 38–45.

[132] K. H. Kim, Y. R. Seong, T. G. Kim, and K. H. Park, "Ordering of simultaneous events in distributed DEVS simulation," *Simul. Pract. Theory*, vol. 5, no. 3, pp. 253–268, Mar. 1997.

[133] P. Peschlow and P. Martini, "Towards an efficient branching mechanism for simultaneous events in distributed simulation," in *Workshop Princ. Adv. Distr. Simul.*, 2006, p. 133.

[134] ——, "A discrete-event simulation tool for the analysis of simultaneous events," in *Proc. Int. Conf. Perf. Eval. Method. Tools*, 2007, pp. 1–10.

[135] ——, "Efficient analysis of simultaneous events in distributed simulation," in *IEEE Int. Symp. Dist. Simul. Real-Time App.*, 2007, pp. 244–251.

[136] M. Schordan, T. Oppelstrup, M. K. Thomsen, and R. Glück, "Reversible languages and incremental state saving in optimistic parallel discrete event simulation," in *Int. Conf. Reversible Comput.*, 2020, pp. 187–207.