RICE UNIVERSITY

Understanding Congestion in High Performance Interconnection Networks Using Sampling

by

Philip A. Taffet

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE Master of Science

APPROVED, THESIS COMMITTEE:

[Signature]
John M. Mellor-Crummey, Chair
Professor of Computer Science and of Electrical and Computer Engineering

[Signature]
T.S. Eugene Ng
Professor of Computer Science and of Electrical and Computer Engineering

[Signature]
Ang Chen
Assistant Professor of Computer Science and of Electrical and Computer Engineering

[Signature]
Ian Karlin
Computer Scientist
Lawrence Livermore National Laboratory

Houston, Texas
April, 2018
ABSTRACT

Understanding Congestion in High Performance Interconnection Networks Using Sampling

by

Philip A. Taffet

The computational needs of many applications outstrip the capabilities of a single compute node. Communication is necessary to employ multiple nodes, but slow communication often limits application performance on multiple nodes. To improve communication performance, developers need tools that enable them to understand how their application’s communication patterns interact with the network, especially when those interactions result in congestion. Since communication performance is difficult to reason about analytically and simulation is costly, measurement-based approaches are needed. This thesis describes a new sampling-based technique to collect information about the path a packet takes and congestion it encounters. Experiments with simulations show that this strategy can distinguish problems with an application’s communication patterns, its mapping onto a parallel system, and outside interference. We describe a variant of this scheme that requires only 5–6 bits of information in a monitored packet, making it practical for use in next-generation networks.
Acknowledgments

First, I would like to thank Prof. John Mellor-Crummey, my long-time research advisor. Many ideas in this thesis came from discussions with him, and the quality of my writing greatly benefited from his feedback. I would also like to thank the other members of my committee: Dr. Ian Karlin, Prof. Eugene Ng, and Prof. Ang Chen. Their perspectives broadened my understanding of my work. I also thank the CODES developers, and especially Nikhil Jain, Misbah Mubarak, and Noah Wolfe for answering my questions about CODES. Thanks to Prof. Alan Cox and Prof. Keith Cooper for encouraging me to stick with research.

Second, I would like to thank the friends I’ve made in my five years at Rice. They have made my life so much happier and have made me a better person. I would especially like to thank Brett Gutstein and Hannah Krusleski for being beside me during much of the writing, literally and metaphorically, and Xavier Gonzalez, Luke Kramer, and Yamin Arefeen for a wonderful weekend in Boston that helped me with a final push to finish my thesis. Thanks to Brett Gutstein and Hunter Tidwell for their helpful feedback at my last-minute practice thesis defense, Tiffany Tang for some pointers to statistics textbooks, and Jake Nyquist for working with me on many projects over the years. Far more than for these favors, I thank my friends simply for their friendship.

Third, thanks to everyone, especially Mary, who prayed for me as the thesis deadline drew near. With the crazy circumstances surrounding the submission of this thesis, it took a miracle for everything to work out.

Finally, I would like to thank my family: Mom, Dad, Amy, Tyler, Lori, Jason, Catherine, Allison, Jonathan, Julia, and Merryn. I dedicate this thesis to them.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vii
List of Tables x

1 Introduction 1
   1.1 Factors impacting communication performance . . . . . . . . . . . . . 2
   1.2 A tool for understanding communication performance . . . . . . . . 6

2 Related work 9
   2.1 Reservoir sampling . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
   2.2 Network congestion . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   2.3 Topologies and routing . . . . . . . . . . . . . . . . . . . . . . . . . . 13
   2.4 Network performance is often unintuitive . . . . . . . . . . . . . . . 14
   2.5 Other approaches for understanding congestion and communication
      performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
      2.5.1 Network simulators . . . . . . . . . . . . . . . . . . . . . . . . 18
   2.6 Profiling techniques for distributed systems . . . . . . . . . . . . . . 19
   2.7 Techniques for improving communication performance . . . . . . . . 20
   2.8 Understanding other aspects of networks . . . . . . . . . . . . . . . 22
      2.8.1 Packet sampling for denial-of-service attribution . . . . . . . 22
      2.8.2 Network tomography . . . . . . . . . . . . . . . . . . . . . . . 25

3 Reservoir sampling for measuring network traffic and
congestion
3.1 Intuition .................................................. 26
3.2 The algorithm .............................................. 27
3.3 Statistical analysis ....................................... 32
3.4 Visualization methodology .............................. 35
3.5 Simulation methodology .................................. 37

4 Case studies and applications 39
4.1 Gaining insight into traffic and congestion ............. 39
4.2 Suggesting a better mapping .............................. 42
  4.2.1 Reverse Cuthill-McKee .............................. 46
  4.2.2 Graph partitioning .................................... 49
  4.2.3 Geometric approach .................................. 51
  4.2.4 Discussion of mapping improvements ................. 54
4.3 Fundamental congestion caused by a communication pattern ... 56
4.4 Congestion due to background traffic .................... 60

5 Implementation considerations and approaches 65
5.1 The current storage requirement .......................... 66
5.2 Probabilistic approach .................................... 67
5.3 Error analysis ............................................... 72
5.4 Hash functions ............................................. 75
5.5 Experimental analysis of probabilistic approach .......... 79
5.6 Evaluating the storage requirements of the probabilistic approach ... 82
5.7 An even smaller variant ................................... 84
5.8 The Bloom filter perspective ............................. 85
5.9 Generating random numbers .............................. 86
5.10 Identifying congestion .................................... 89
6 Conclusion and future work

6.1 Future directions

Bibliography
Illustrations

3.1 A sample congested fraction plot. ................................................................. 36

4.1 Congested fraction plot over the full execution of the 2-D stencil pattern using a row-major mapping. ................................................................. 43
4.2 Congested fraction plots during the $+y$ phase of the stencil communication pattern with the row major mapping. ................................. 44
4.3 A small portion of the congested fraction plot using only traffic received by compute node 1446 during the entire execution, with one path used during the $+y$ phase annotated. ................................................ 45
4.4 The adjacency matrix of the communication graph for the 2-D stencil pattern with the row major mapping (black) and the reverse Cuthill-McKee mapping (red). ................................................................. 47
4.5 Congested fraction plot of the 2-D stencil pattern with a reverse Cuthill-McKee mapping. ................................................................. 48
4.6 Congested fraction plot of the stencil communication pattern using a mapping produced with the Metis partitioner. .................................. 50
4.7 Congested fraction plot of the stencil communication pattern using a mapping derived from our incremental graph partitioning algorithm. .... 50
4.8 A non-rectangle unit-tileable shape with area 5 and perimeter 10. ........ 53
4.9 Congested fraction plot of a stencil communication pattern using a tiled mapping. ................................................................. 54
4.10 Congested fraction plot of the stencil communication pattern using a random mapping. .......................................................... 55

4.11 Chronological progression of congested fraction plots of five sub-intervals of the execution of the reduction-like pattern to rank 0. Time progresses from top to bottom and is labeled on the left of each image in milliseconds based on packet arrival time. ............... 58

4.12 Congested fraction plot of the naïve reduction pattern to rank 0. .... 59

4.13 Congested fraction plot of the naïve reduction pattern to rank 388. .. 59

4.14 Congested fraction plot of the primary application (ring exchange) running in isolation on 2172 nodes distributed throughout the system. 61

4.15 Congested fraction plot of the background traffic (uniform random traffic) running in isolation on 2436 nodes distributed throughout the system. ................................................................. 61

4.16 Congested fraction plot from the perspective of the primary application while running simultaneously with the background traffic application. ................................................................. 62

4.17 Congested fraction plot from the perspective of a network administrator while the primary application runs simultaneously with the background traffic application. ......................................... 63

5.1 The values of $H(id, d)$ for the specified hash functions, with a fixed randomly chosen $id$, and $d$ ranging from 1 to 900, numbered left to right, top to bottom. ................................................................. 76

5.2 Congested fraction plot generated using the probabilistic variant of our reservoir sampling-based scheme of the naïve reduction to rank 0 with 6.5 million packets. ................................................................. 80
5.3 Congested fraction plot generated using the probabilistic variant of our reservoir sampling-based scheme of the naïve reduction to rank 0 simulating 10 times as many packets but the same total communication volume. ................................. 80

5.4 Congested fraction plot generated using the optimized probabilistic variant of our reservoir sampling-based scheme of the naïve reduction to rank 0 with 6.5 million packets. ................................. 81

5.5 Congested fraction plot generated using the optimized probabilistic variant of our reservoir sampling-based scheme of the naïve reduction to rank 0 simulating 10 times as many packets but the same total communication volume. ................................. 82

5.6 A 3-bit shift register that will output 0 1/3 of the time. ................. 88

5.7 A circuit that generates an output of 0 approximately 1/l of the time. 88
Tables

3.1 A comparison of the simulated networks used in this thesis . . . . . 38

4.1 A comparison of the different mappings used with the 2-D stencil
pattern. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
Chapter 1

Introduction

For several decades, exploiting parallelism has been an essential strategy for efficiently solving large and computationally expensive problems. In tightly-coupled applications, as many cores simultaneously compute different portions of a problem, they must communicate to share data. In modern high performance computing systems, this communication happens at many levels: cores on the same chip communicate with each other using an on-chip network, such as the on-die ring network in Sandy Bridge CPUs [1]; chips within the same compute node communicate with each other using an on-node network, such as Intel’s QuickPath [2]; and nodes communicate with each other using a system-level interconnection network. Although all levels are important for overall performance, this thesis focuses on communication over system-level interconnection networks, or interconnects. This type of interconnection network connects compute nodes to each other and to other resources such as a parallel file system. This thesis focuses on indirect networks, i.e., networks where endpoints such as compute nodes are directly connected to routers or network switches but not to each other.

The performance of communication over the interconnect can have a significant impact on the overall performance of applications. Additionally, sensitivity to communication performance often increases as the scale of an application increases, so efficient communication is especially important to scalable parallel applications running on large-scale parallel systems.

This work develops a technique for measuring various aspects of communication
performance with the end goal of accelerating applications by understanding the underlying causes of any slow communication.

1.1 Factors impacting communication performance

Most parallel HPC applications communicate through a library that implements the Message Passing Interface (MPI) [3], and so these applications are ultimately exposed to the performance of the network almost entirely through how long calls to various communication routines take to complete. To understand the factors that impact communication performance, we consider some of the steps involved in a simple point-to-point communication:

- First, some setup work is performed, and the data to be sent is transferred to the network interface card (NIC), which connects the compute node to the interconnect.* The NIC splits the message and encapsulates it in a packet.

- When the NIC is able to begin sending a packet, it sends it to the switch to which it is connected.

- After receiving the packet header, the switch makes a routing decision to determine the packet’s next hop. As the switch receives the packet, if the next hop is able to receive the packet, then the switch forwards the packet in pieces as it arrives. If the next hop is not able to receive the packet, the switch temporarily stores the packet in a buffer.

- The previous step repeats until all packets in the message reach the recipient.

---

*This work mostly ignores these on-node steps, although they can also have significant impact on performance [4].
Finally, the NIC of the recipient provides the data to the application through the communication library.

Since interconnects are designed for high throughput and low latency, they are typically lossless and employ credit-based flow control and explicit congestion notification (both discussed later). Examples include InfiniBand [5], Intel Omni-Path [6], and the forthcoming Gen-Z [7] protocol. In contrast, Ethernet is lossy, which means switches can drop packets in response to congestion, which shapes the flow control and congestion control techniques it uses. Although InfiniBand, Omni-Path, and Gen-Z are not identical, the protocol aspects that are most important for this work are present in all three protocols. Additionally, each protocol gives a different name to network endpoints, for which we use the generic term NIC: host channel adapter in InfiniBand, host fabric interface in Omni-Path, and requester/responder in Gen-Z. Most of the rest of this thesis applies to all three protocols, but when a specific choice is necessary, we focus on InfiniBand as an exemplar of this type of network.

An important aspect of InfiniBand is that it is typically lossless. This means that under normal operation, compliant networking hardware cannot drop messages; every packet that is received must be immediately forwarded or stored until it is forwarded. Since any buffer in a switch can only store a finite amount of data at a time, maintaining losslessness means that care must be taken to ensure a switch has space in its buffer for data it receives. Buffer space is managed using a credit-based flow control scheme [8], which enforces the policy that a switch is not allowed to send more bytes than can fit in the available space in the destination switch’s buffer. A switch must wait for sufficient credit, i.e. space in the destination switch’s buffer, before sending any traffic. This waiting increases latency, decreases bandwidth, and increases the time it takes to complete the communication routine. The term network
congestion, a major focus of this work, primarily refers to the impact of this waiting. Additionally, the latency of a message depends heavily on the path taken from the sender to the receiver. In the absence of congestion, a path with more hops will take longer to transmit a packet. Modern networking hardware, such as Omni-Path switches [6], employs adaptive routing; switches’ routing decisions are based on the current state of the network [9]. Adaptive routing algorithms are designed to reduce congestion by routing packets along less congested paths. However, adaptive routing with incomplete state information is imperfect and poor routing choices decrease communication performance.

At a higher level, communication performance is also impacted by which messages need to be transmitted, and where. These are determined by the communication pattern and the mapping from logical parallel tasks to physical processors. HPC applications typically communicate using a logical topology that matches the problem the application is solving. For example, a three-dimensional stencil code may use a three-dimensional grid as a logical topology [10]. An application’s communication pattern specifies the source and destination of each message at the level of the logical topology. A three-dimensional solver might use a 6 point stencil communication pattern, where each node communicates with its top, bottom, left, right, front, and back neighbors in the logical topology. In spite of the convenience of specifying communication partners at the level of the logical topology, real packets travel between physical compute nodes, not logical tasks, so the mapping between an application’s logical topology and the network’s physical topology, called the communication mapping, is important. The interaction between the communication pattern and communication mapping impacts the amount of congestion and path lengths.

Finally, the design of the network itself impacts communication performance, but
unlike other factors, the network is mostly fixed once purchased and installed. Many modern HPC interconnects are based on either fat-tree or dragonfly topologies, or variants [11, 12]. These topologies admit a number of variations, such as multi-rail or multi-plane fat trees [13], and different link widths or speeds. The design of the network affects almost every aspect of its performance, especially its sensitivity to congestion. One strategy to improve the performance of an interconnect is to knowingly over-provision it at procurement time, but the obvious downside to this strategy is the cost. This cost-performance trade-off, known as “right-sizing the interconnect”, is important for the overall effectiveness of the HPC system. Interconnects in large HPC systems that are currently deployed may be over-provisioned [14, 15]. Careful choices in the design of future interconnects could result in significant cost reductions and only a small performance penalty. Since the procurement could reinvest savings on more powerful compute nodes, these choices could also result in an overall total performance increase.

The impact of communication performance on overall performance varies between HPC applications and increases as an application scales to more compute nodes. Surveys of some collections of HPC benchmarks and applications [16, 17] show that of 14 applications, 6 spend < 10% of the execution time in communication, 4 spend around 20%, 2 spend around 40%, and 2 spend > 50% total execution time in communication. This is not a comprehensive survey, but it shows that many applications do spend a significant amount of execution time communicating, which means that their overall performance can be noticeably impacted by changes in their communication performance.
1.2 A tool for understanding communication performance

Modern CPUs have precise, fine-grained counters that measure many aspects of performance, including an application’s interaction with a node’s memory subsystem. Performance analysis tools can use these counters to measure work, resource consumption, and inefficiency, providing HPC application developers with actionable insights into the performance of their application. First party tools like HPCToolkit [18] can even provide this information in a scalable and minimally invasive form by sampling the counters.

The state of the art for network performance analysis lags in comparison. Recording comparable information about network performance requires combining information from the application and networking hardware spread all over the network. Additionally, although some switches have performance counters [19], they are designed to be read with fairly low frequency compared to CPU counters. For example, by default, the Fabric Manager for Intel Omni-Path switches collects performance information every 10 seconds [20]. This is frequent enough for diagnosing overall health problems with the network (e.g., a failing link), but not for attributing performance information back to specific lines of codes or methods in an application.

Correlating between data about a packet provided to a central fabric manager-like application by different switches is difficult but not intractable. If the path from the switches to the fabric manager is shared with the application’s data, recording performance metrics frequently could distort application performance. Furthermore, data at the fabric manager is not attributed to an individual application or batch job, adding an additional step before processing.

The natural solution of storing the performance data in the packet eliminates the need for a separate correlation step but comes with its own challenges. Switches in
high performance interconnects tend to use cut through switching to reduce latency which essentially precludes adding additional headers in flight. Thus, a method of gathering performance information while only using a fixed and very small amount of storage is desirable.

Current communication performance analysis tools and techniques do not deliver sufficiently rich information for application developers to understand fully their application’s communication performance. Overall communication performance depends on many factors— including the communication pattern and its mapping to the physical network topology, as well as the presence of background traffic—and complicated interactions between them, so measuring communication performance-related phenomena is a powerful alternative to extremely detailed modeling.

**Thesis statement:** A distributed network monitoring scheme based on reservoir sampling that samples the links through which a packet passes and the congested links the packet encounters along its route provides a practical and powerful approach for quantitatively understanding network traffic and network congestion. The proposed technique uses a constant amount of storage per packet and captures both network-wide and time-varying behavior. Simple rules presented in this thesis guide an application developer from the quantitative measurements collected with this approach to a diagnosis of the communication performance problem when it is in one of three categories: a poor communication pattern, a poor mapping onto the network topology, or background traffic from other applications. A useful variant of the technique can be implemented using as few as five to six bits of information inside a monitored packet and requires only lightweight support on individual network switches, which makes it worth considering for inclusion in next-generation networks, where it can give application developers valuable insight into the communication
performance of their HPC applications.

The rest of the thesis is as follows: Background and related work are discussed in chapter 2. In chapter 3, we explain our distributed network monitoring scheme based on reservoir sampling. We evaluate the scheme with several experiments in chapter 4 and explain how an HPC application developer can use the measurements our scheme collects to diagnose communication performance problems. We describe variants of the scheme that are more practical for hardware implementation in chapter 5, along with some exploration of other implementation considerations. Finally, chapter 6 summarizes our conclusions and future work.
Chapter 2

Related work

2.1 Reservoir sampling

Reservoir sampling is a family of statistical techniques for efficiently choosing a fixed-size random sample from a large population, perhaps of unknown size. The first such algorithm, known as Algorithm R (see algorithm 1), is due to Waterman [21]. This algorithm cleverly combines two decisions—should the current record be selected to enter the reservoir? If so, what record in the reservoir should it replace?—with one random integer generated on line 5. Notice that the first decision happens with probability $n/t$ for the $t^{th}$ record, where $t > n$. The record in the reservoir to replace with the $t^{th}$ record is chosen uniformly from the $n$ records in the reservoir (i.e., given that $M < n$, each has a $1/n$ chance of being replaced).

Vitter gives a more detailed analysis of Algorithm R and provides several more efficient algorithms [22]. The efficiency gains in these algorithms come from skipping more than one record at a time, which is useful when there are many records and skipping several records at a time is cheaper than reading each record, e.g., if the records are read from a hard drive. However, when $t$ is fairly small, the number of records to skip will also be small, limiting the gains from using Vitter’s algorithms.
Algorithm 1: Algorithm R: Reservoir Sampling

**Input:** A positive integer $n$ indicating the size of the desired random sample;

A stream $A$ of records of arbitrary length (at least $n$)

**Output:** $n$ records chosen uniformly randomly from $A$

1. Copy the first $n$ records of $A$ to a new array reservoir

2. $t \leftarrow n$

3. while $A$ has more records do

4. \hspace{1em} $t \leftarrow t + 1$

5. \hspace{1em} $M \leftarrow$ a random integer chosen uniformly in the range $0 \leq M \leq t - 1$

6. \hspace{1em} if $M < n$ then

7. \hspace{2em} reservoir[$M$] $\leftarrow$ next record from $A$

8. \hspace{1em} else

9. \hspace{2em} Skip next record from $A$

10. return reservoir
2.2 Network congestion

Pfister and Norton’s seminal work [23] introduces the idea of tree saturation on interconnection networks. No matter the bandwidth or topology of the network, each compute node’s ability to receive information from the interconnect has a finite bandwidth limit, which we refer to as the ejection bandwidth (the symmetric value to injection bandwidth). Thus, for a sufficiently large network, more than two nodes in the common case, the sum of the injection bandwidth of all nodes except one exceeds the ejection bandwidth of one node. This occurs in particular with “hot spot” patterns, where many nodes send a significant amount of traffic to a single destination, as in the case of heavy contention for a naïve implementation of a global lock or a parallel file system. Unless the interconnect itself does not conserve flows (e.g., using message combining), the buffers in the switch connecting to the hot spot node become saturated. This switch is then not able to receive messages at its normal rate, causing congestion in the switches that connect to it. One of the key insights in the paper is that this tree of congestion also slows down messages with destinations outside of the hot spot, simply because the route to the destination passes through a switch that is part of the tree of congestion. Garcia, et al. also summarize Pfister and Norton’s findings and later research to explain tree saturation, congestion, and related concepts [24].

Several decades later, Pfister revisited the issue of network congestion [25]. In particular, Pfister evaluates the InfiniBand Congestion Control architecture, claiming that it is the first effective solution to tree saturation. The InfiniBand Congestion Control architecture is based on the scheme developed by Ramakrishnan and Jain called explicit congestion notification [26]. Switches that are congested set to 1 a single bit in the header called the congestion indication bit, or forward explicit con-
gestion notification (FECN) bit. Switches that are not congested ignore the bit, and, in particular, they do not set it to 0. Since the bit is initialized by the sender to 0, the receiver knows if the packet encountered any congestion. It notifies the sender by copying this bit into the acknowledgement for that packet. Upon receiving the acknowledgement marked with the congested bit, now referred to as the backward explicit congestion notification (BECN) bit, the sender reduces its sending rate. The InfiniBand specification [5] contains the strategy used to reduce the sending rate, and eventually re-increase it when the congestion has subsided, but the exact parameters are left to vendors and system owners. Choosing these parameters optimally is difficult, and Pfister, et al. were the first to show that there is a choice of parameters, at least for one simulated 128-node fat tree network, that works effectively [25].

In an effort to address the parameter-setting difficulties with the InfiniBand Congestion Control architecture, Liu and Russell propose a slight modification called Red and Green lights-Based Congestion Control [27]. This scheme temporarily pauses senders when a packet with the BECN bit set is received. When the congestion in a switch has subsided beyond a second threshold, the switch notifies clients who may have received a BECN bit, allowing them to resume sending. As a minor point, to handle multiple simultaneous hot spots, this scheme requires any switch that sets the FECN bit in a packet also to copy its switch ID into that packet.

The problem of network congestion certainly applies outside of specialized interconnection networks for HPC systems. Commodity IP routers typically drop incoming packets when the buffers are full, and there are several TCP congestion control protocols that react to this indication of congestion. One specific example is TCP Random Early Detection (RED) [28], which probabilistically drops packets prior to running out of buffer space. These probabilities are based on the smoothed queue
lengths, so that as congestion increases in the router, the probability of dropping a packet increases as well. Unfortunately, the initial version of TCP RED had some issues, and was never fixed or widely adopted.

2.3 Topologies and routing

Additional efforts to counter network congestion include switching to high-radix topologies and a heavy reliance on adaptive routing. The dragonfly topology [29] is an example of a topology that, at a lower cost than a comparable fat-tree, provides a large amount of bisection bandwidth and routes uniform traffic very efficiently. The trade-off is that it does not handle all common communication patterns well. Although there are many paths connecting a source to a destination (i.e., high path diversity), not all paths are minimal, unlike in a fat tree. Hence, dragonfly networks are more sensitive to routing decisions than some other topologies, so a well-performing adaptive routing scheme is necessary to ensure high performance. Unfortunately, theoretically optimal routing algorithms require global knowledge of the network to make a local decision, which is very difficult to implement in practice. Instead, algorithms like the local variant of Universal Globally-Adaptive Load-balanced routing [30, 29] are used in practice. The complexity of this issue is exacerbated by the fact that the adaptive routing algorithm’s performance depends in part on the workload [31].

Other common network topologies include torus and fat tree topologies [32]. Computational problem domains that naturally decompose into a grid map naturally onto a system with a multi-dimensional mesh or torus network, but mesh and torus topologies tend to have large network diameters, leading to poor performance for some communication patterns. An interesting difference between torus networks and other networks is that the network switching and computation are integrated into the same
Fat tree networks conceptually consist of a tree of network switches with compute nodes connected to the switches at the leaves of the tree. In a full bisection bandwidth fat tree, link bandwidth increases towards the root of the tree so that each switch has equal aggregate bandwidth in links toward the root as in links away from the root. This ensures that the cumulative bandwidth between the two partitions in any equal partitioning of the compute nodes does not depend on the choice of partitioning. Additionally, full bisection bandwidth fat trees can route any traffic pattern almost as efficiently as the optimal topology for that pattern built with the same networking hardware [33].

Directly implementing the conceptual description of a fat tree requires individual links with impractically large amounts of bandwidth. Thus, for practical reasons, fat tree networks are typically built with several switches acting as one conceptual switch at levels closer to the root, and the “fat” links are implemented as several individual links to different switches.

A common variant of this topology is the fractional bisection bandwidth fat tree, also commonly called tapered fat tree. Tapered fat trees sacrifice the full bisection bandwidth property to reduce the cost of the network. One or more levels of the tree network has more aggregate bandwidth away from the root than towards the root of the tree.

2.4 Network performance is often unintuitive

Because so many factors impact network performance, as discussed in section 1.1, application developers typically use intuitive rules of thumb. For example, to ensure good communication performance, one should:
(i) minimize the number of bytes transmitted through the network,

(ii) minimize the distance (number of hops) each message must travel, and

(iii) overlap communication with computation.

While these rules of thumb are reasonable, and simple enough to reason about, unfortunately, they are occasionally misleading, especially rule (ii). Following these rules blindly can hurt performance, as several more careful studies have shown.

Bhatele, et al. observe that that using a tilted mapping on a torus improved communication performance [34]. Tilted mappings increase the distance messages travel, but have higher path diversity. They found that for large messages, the benefit of this path diversity, which is harder to reason about and does not appear at all in the rules of thumb, outweighed the small increased in latency.

Similarly, on a two-level network like dragonfly, maximizing locality (i.e., following rule (ii)), leads to hot-spots [35]. For the 2D stencil, 4D stencil, and multicast communication patterns, adding some randomness to the task mapping improves performance in spite of also increasing total traffic and increasing the distance each message travels.

Later work [36] explores the nuances of this finding, especially when multiple jobs are involved. When communication-intensive jobs (bullies) and less communication-intensive jobs run simultaneously with random placement and adaptive routing, the performance of the less communication-intensive job suffers (i.e., it is bullied). In this case, using a contiguous placement policy, which improves locality, limits the impact of the bully and improves performance. In this case, rule (ii) holds, but for much more complicated reasons.

Some of our previous work [14] shows that rule (i) has problems too. On several
systems at Lawrence Livermore National Lab (LLNL), sending a large message from one MPI rank to an MPI rank on a different node can be faster than sending a message to a rank on the same node. In this case, trying to minimize the number of bytes transmitted through the network by keeping more data within the node hurts performance.

The other aspect of rule (i), i.e., reducing the total amount of communication, is the cornerstone of communication-avoiding algorithms, and is more sound.

The main difficulty with rule (iii) is that not all MPI libraries support actual overlap by default. Code that uses non-blocking MPI calls may appear to overlap communication with computation, but the communication may not happen until MPI::Wait is called.

### 2.5 Other approaches for understanding congestion and communication performance

Yoga and Chabbi attempt to tackle essentially the same problem as this thesis [37]. Like this work, they develop a performance monitoring scheme that tracks individual packets. This enables CPU-side context, especially call paths, to be correlated with network performance measurements to deliver end-to-end insights. To reduce the volume of performance measurement data, they only collect data about packets marked with the Performance Monitoring (PM) tag in the Gen-Z protocol [7, Section 8.42]. Since the PM tag is set by the source NIC, this enables the sender of the message to control the sampling procedure. In their scheme, this performance data is aggregated at a collection server, which has enough information to reconstruct the full path of any marked packet. Additionally, since the performance measurement data collected
by the switches contains timestamps, when aggregated, this scheme provides rich
information about how long each step in the network took.

The Ohio State University’s INAM² tool [38] also aims to monitor communication
performance in the context of an application and the topology of the network. INAM²
correlates data from a batch scheduler, switch performance counters, and metrics
provided by an MPI runtime. It generates a network map with links color-coded
based on traffic, and it can track routes through a specific link. Unfortunately, due
to the poor granularity of the data provided by their data sources, the tool can only
identify the process that generated traffic and whether the traffic comes from point-
to-point operations, collective operations, or RDMA accesses; it cannot track traffic
back to individual MPI calls. Additionally, the tool’s support for dynamic routing
is somewhat poor, it does not specifically measure congestion, and like other switch
counter-based tools, it has poor time granularity.

Mellanox includes some congestion monitoring capabilities in the Unified Fabric
Manager software [39]. Unfortunately, Mellanox has published very little about the
congestion mapping features, but based on available information, it appears to ag-
gregate congestion levels based on tiers in the network. It also features job-level
aggregation of traffic and congestion statistics with 30 second granularity that can be
increased to two second granularity in a special mode [40]. Congestion is measured by
computing the fraction of theoretical bandwidth that is utilized. The Unified Fabric
Manager is designed for a system administrator as opposed to an application devel-
oper, so it focuses on corrective actions like optimizing the network’s routing policies.
Like this work, the Unified Fabric Manager focuses on distinguishing between several
causes of congestion: routing, job placement, or resource allocation inefficiencies [41].

Vetter proposes PHOTON [42], a lightweight MPI profiling tool. PHOTON adds
to the header of each sampled MPI message a timestamp indicating the send time and a source code location identifier. When the message arrives, the tool can use the fields to compute message latency and to attribute it to the send program context and the receive program context. Vetter also considers several methods for sampling messages and determines that random sampling of each message is effective.

2.5.1 Network simulators

Network simulators can be a valuable tool for understanding communication performance. An application’s communication behavior can be heavily instrumented without disturbing execution and can be replayed several times in a controlled environment. For this work, we use the CODES simulation framework [43]. CODES is a high-performance, parallel, flit-level simulator that specializes in simulating interconnects for HPC systems. CODES is built on ROSS, a parallel discrete event simulator.

Although CODES does not specifically implement the InfiniBand or Omni-Path specifications, it models an input-queued and output-queued switch that uses credit-based flow control. The developers of CODES emphasize validation of network models to ensure their accuracy [44].

CODES can be driven in several ways, including with synthetic traffic and with traces. Synthetic traffic is generated by a small application that determines the destination of each message, given a source rank number. Traces of normal MPI applications can be collected with ScoreP [45] and replayed with TraceR [46]. Synthetic traffic applications are primarily point-to-point, while TraceR also supports replaying MPI collectives.

One deficiency in the fidelity of CODES is that it doesn’t implement explicit
congestion notification (see section 2.2). Because of this, CODES exhibits parking lot unfairness [9].

2.6 Profiling techniques for distributed systems

Just as collecting end-to-end information is helpful for diagnosing network performance issues, it is also valuable for debugging large distributed systems because it includes more context. X-Trace [47] and Dapper [48] both use unique task identifiers to correlate activity between distributed tasks involved in processing a request. X-Trace and Dapper transmit task identifiers in-band but transmit larger trace data out-of-band. In Dapper, this enables buffering of trace data to reduce the performance overhead and collecting trace data after the primary result has been returned. X-Trace also claims this makes it possible to collect trace data from requests that fail.

Whodunit [49] collects end-to-end information with finer granularity, focusing on tracking operations through shared memory as well as other events. For certain stages, Whodunit stores the entire operation context with the operation, but for transactions that cross nodes, it uses a 4-byte identifier similar to the task identifiers in X-Trace.

All three tools are primarily concerned with generating diagrams that illustrate the interactions between several parts of the system. The focus of these tools on computation means that if they collect networking performance information at all, they treat the network as a black box.

In contrast to normal CPU profilers that sample CPU events, ProfileMe [50] samples instructions. It collects several pipeline stage latencies and whether certain events occurred for a single instruction as it executes. This enables precise attribution of events to the instruction that caused the events, even if the CPU uses complicated
out-of-order dispatch and retirement schemes. Additionally, ProfileMe can collect information about pairs of instructions, which provides detailed information about the instruction-level parallelism exposed by each instruction. ProfileMe tags instructions to be profiled with a small identifier, unique only among profiled instructions simultaneously in flight, and stores the latency and event information in on-chip registers. The CPU periodically delivers an interrupt to the operating system and provides the recorded performance information.

### 2.7 Techniques for improving communication performance

As mentioned in chapter 1, an application’s mapping between its logical communication topology and the physical topology of the network on which it runs impacts communication performance. Both topologies can be naturally represented as graphs with weighted edges; edge weights in the logical topology graph correspond to communication volume between two logical processing elements, and edge weights in the physical topology graph correspond to link capacities. Hoefler and Snir [51] use two metrics to characterize the performance of a mapping between these two graphs: 

- **average dilation** (equivalent to *hop-bytes*)
- **worst case congestion**, which is defined as 
  \[
  \text{max}_{e \in \text{edges}} \frac{\text{communication volume through link } e}{\text{bandwidth of link } e}.
  \]

They prove that the problem of finding a mapping that minimizes either metric is NP-complete. They evaluate several heuristics for finding an approximate solution to the mapping problem in polynomial time: a greedy algorithm, a recursive bisection graph partitioning-based algorithm, and graph similarity-based techniques. They find that all three techniques reduce worst case congestion and improve communication performance on a sparse matrix-vector multiplication benchmark.

Bhatele’s PhD thesis [52] also explores the mapping problem for a number of
applications and network topologies. After demonstrating speedups of over 50% by designing a mapping specifically for an individual application and several network topologies, he presents a new automatic mapping framework. His framework uses pattern matching, and it works on both regular and irregular communication graphs.

Additionally, Srepathi, et al. [53] explore several automated techniques for mapping several applications onto a 3D torus network. They use a graph partitioning technique, the reverse Cuthill-McKee algorithm (both described in section 4.2), and a tree-based clustering algorithm. They find that for the majority of the 11 applications they consider, at least one mapping performs better than the default mapping. However, the most effective mapping improvement technique varies between applications.

Because these techniques are all static, they cannot capture the full dynamics of application communication. If two paths with heavy traffic do not overlap in time, they can overlap on the physical network without decreasing performance. Algorithms that only use aggregate communication volume cannot take advantage of this fact.

On the other hand, Kamil, et al. [54] explore theoretical alternative network designs that would allow modifying a physical network topology to suit an application’s logical communication topology. While promising, these designs require new special purpose switching hardware, which makes them likely impractical in the short- or medium-term. Dynamically reconfigurable networks have been explored in greater depth in datacenter-type environments [55].
2.8 Understanding other aspects of networks

2.8.1 Packet sampling for denial-of-service attribution

Surprisingly, the literature on tracing denial of service attacks on the Internet, a problem known as IP traceback, contains approaches closely related to the technique developed in this thesis. Savage, Wetherall, et al. describe a technique to trace the path of packets through the Internet [56]. The primary application of this technique is to determine the source of IP packets in a denial-of-service attack, even when the attacker has full control over the packets, and in particular, can spoof the source address field. They suggest two sampling-based strategies.

The first, simpler strategy samples routers along the route. Each router replaces a special node field with its own IP address with a fixed probability $p$. The victim can deduce the packet’s path based on the number of times each IP address appears. Approximately $p$ packets that the victim receives should contain the IP address of the last router along the packet’s path. Of the $1 - p$ packets not marked with the last router’s IP address, $p$ should be marked with the IP address of the second to last router. In general, the $k^{th}$ to last router’s IP address appears in approximately $p(1 - p)^{k-1}$ of the packets. One strength of this technique is that the router’s behavior is not based on information in the packet, which may be spoofed by the attacker. The attacker can spoof the node field, but only in such a way that adds fake routers to the far end of the path that the victim deduces. The biggest downside of this technique is that the number of packets needed to recover a complete path grows exponentially with the length of the path.

The second strategy samples links along the route. With probability $p$, a router replaces both the start field with its IP address and the distance field with 0. The
subsequent router, on observing $distance = 0$, sets the $end$ field to its IP address. Any router that does not reset $distance$ also increments it, which tells the victim how far away the sampled link was. This also ensures that an attacker is only able to spoof edges beyond the edge of the actual path.

Additionally, this paper contains several potential modifications to reduce the number of bits required:

- Store the exclusive or of $start$ and $end$ in a single field.

- Split IP addresses into $k$ fragments and randomly choose which fragment to use as an identifier. As long as the fragment index is also specified, it is possible to reconstruct the full IP address in $O(k \log k)$ times as many packets.

These modifications are in the same spirit as some of the modifications suggested in chapter 5.

Song and Perrig [57] propose some modifications to this packet marking scheme that assume the victim of the attack has a graph of the upstream routers. This reduces the number of false positives and the computational cost when there are several attackers. They also propose using a family of hash functions to fit an IP address in an 11-bit field in the IP header they hope to overload while reducing the impact of hash collisions. This is equivalent to splitting a large hash (possibly larger than the IP address) into several fragments, but unlike the second modification in Savage, et al., when combined with a network graph, it allows incremental reconstruction of

---

*This reduces the number of bits needed to store an IP address in the header from $N$ to $\lg k + N/k$. The minimum value is attained when $k = N \log 2 \approx N/2$. This means a 32-bit IPv4 address can be transmitted with 6 bits in each packet (4 for the fragment index and 2 for the fragment value) and a 128 bit IPv6 address can be transmitted with 8 bits in each packet.*
the attack paths. Song and Perrig also propose two schemes for cryptographically authenticating the marks routers add to detect spoofing. Our work trusts the networking equipment in an HPC environment, so this is not a concern for this thesis, but it is valuable for IP traceback.

In later work, Yarr, Perrig, and Song [58] solve a slightly different problem: rather than reconstructing the entire attack path, they want to obtain an identifier that represents the path to recognize packets travelling along the attack path. They propose concatenating a few bits of a hash of the IP addresses of the endpoints of each link. They select the bits to store based on the link’s position in the path, which means that even with lots of packets, they may not be able to identify individual links, but their scheme effectively computes a hash of the path each packet takes. Packets with the same identifier as attacking traffic likely travelled along the same path as attacking traffic, and so can be blocked. For performance analysis, it is helpful to know the links along the congested path instead of just an effectively arbitrary identifier of the path.

Snoeren, et al. [59] take a different approach to the IP traceback problem and attempt to reconstruct the entire attack path using a single packet. Rather than storing information inside a packet, they propose adding Bloom filters to each router to store the packets that passed through the router in a given window of time. A victim can ask network operators to query the Bloom filters across their networks to see if a given packet from a denial of service attack passed through each router.

Sattari analyzes several IP traceback schemes in her Master’s thesis [60]. By casting the problem as a coupon collector’s problem (i.e., collecting at least one packet tagged with each hop in the path), she derives that the optimal set of marking probabilities amounts to reservoir sampling of the routers or links along the path.
She also proposes an alternate scheme that stores linear combinations of IP addresses in packets. When a router decides to replace the value in the packet, it instead stores a linear combination of its IP address and the value stored in the packet. Packets also contain a vector of the randomly chosen coefficients used in the linear combination so that the victim of the attack can reconstruct the path by solving a system of equations. This scheme can reconstruct more of the path with fewer packets compared to the scheme that directly replaces the IP address. She compares this scheme to an algebraic approach for IP traceback [61] that also stores linear combinations of IP addresses, but uses the packet ID instead of randomly generated coefficients. She finds that the packet ID-based scheme requires more storage in the packet because the coefficients must be larger.

2.8.2 Network tomography

The field of network tomography [62] centers around the problem of inferring information about networks in situations where it cannot be measured directly, e.g., the Internet. There are two performance-related forms [63]: inferring per-link delays based on end-to-end transmission times, and inferring the quantity of data transmitted between each pair of senders and receivers based on per-link packet counts. Both forms can be analyzed as inverse problems, which have been extensively explored in other fields, and solution techniques exist to estimate the desired parameters. Unfortunately, network tomography does not handle adaptive routing well, and is designed to measure behavior that changes less quickly than on a high performance interconnection network.
Chapter 3

Reservoir sampling for measuring network traffic and congestion

Our overall goal is to provide information to HPC application developers that helps them improve the communication performance of their applications. We believe this demands a quantitative assessment of overall traffic and congestion. This chapter explains our technique—first at a high level, then in depth—that uses reservoir sampling to collect the information we need to deliver this assessment. This is followed by a statistical analysis of the technique, and a description of our evaluation and visualization methodology.

3.1 Intuition

Reservoir sampling, as explained in section 2.1, is a powerful tool for collecting a fixed sized sample from a variable sized pool. However, it’s not immediately clear from what pool we should draw samples to measure congestion. This work proposes using reservoir sampling to select one or more hops along the path a packet takes. We store a reservoir of size $n$ using a small, fixed amount of space in the packet itself. Since the result is the state of the reservoir after all samples have been considered, the recipient of the message can determine the hop or hops selected by simply reading the state of the reservoir from the packet upon receipt.

Intuitively, we can use reservoirs of samples from many packets to determine which switches are handling lots of traffic, but simply handling lots of traffic does not ne-
cessarily imply congestion. Thus, we extend our scheme by adding a second reservoir in the packet to sample the subset of the packet’s hops that are congested at the time the packet is processed. At a high level, capturing samples of congested links in addition to samples of links merely carrying traffic lets us distinguish between different causes and impacts of congestion, as explained in section 4.1. For the rest of this thesis, we fix $n = 1$ and claim this captures enough information about communication performance to be useful.

### 3.2 The algorithm

**Algorithm 2:** Reservoir Sampling for Understanding Congestion: On the sender NIC

**Input:** The current outgoing packet, $packet$

1. $packet.hop\_reservoir \leftarrow (\text{no switch ID, no port ID})$
2. $packet.hop\_count \leftarrow 0$
3. $packet.congested\_reservoir \leftarrow (\text{no switch ID, no port ID})$
4. $packet.congested\_count \leftarrow 0$

There are a few points to note. First, notice the difference between algorithm 3 and algorithm 1. These are in part due to the specialization $n = 1$, and in part to allow for a stream of records of size $< n$ (i.e., empty). We relax the constraint that the reservoir always contain a valid sample so that we don’t need to pre-fill the reservoir with the first $n$ records as in the original algorithm; a first congested link may not be available. Note, however, that when a packet arrives, a reservoir (hop or congested) contains (no switch ID, no port ID) if and only if the hop count is 0. Since the only random integer between 0 and 0 inclusive is 0, the reservoir is always updated when
Algorithm 3: Reservoir Sampling for Understanding Congestion: On every switch

Input: The current packet, `packet`

1. `outport ←` the port through which `packet` will leave the current switch
2. `rand1 ←` a random integer between 0 and `packet.hop_count`, inclusive
3. if `rand1 = 0` then
   4. `packet.hop_reservoir ← (my switch ID, outport)`
   5. `packet.hop_count ← packet.hop_count + 1`
4. if `outport is congested` then
   5. `rand2 ←` a random integer between 0 and `packet.congested_count`, inclusive
   6. if `rand2 = 0` then
      7. `packet.congested_reservoir ← (my switch ID, outport)`
      8. `packet.congested_count ← packet.congested_count + 1`

the count changes from 0 to 1, and thus the reservoir is in a valid state whenever the count of hops or congested hops is greater than 0. Additionally, adding 0 to either `counts` or `congested_counts` in algorithm 4 would have no effect, so the algorithm simply ignores those operations, which would involve reading an invalid reservoir.

Also note that hops are recorded as a tuple containing the output port in addition to the switch ID. Although it is somewhat common to describe a switch as congested or not congested, this is normally not accurate. Even when one path through a switch is congested, many types of switches can continue to forward packets freely along independent paths.
**Algorithm 4:** Reservoir Sampling for Understanding Congestion: On the receiver NIC

**Input:** The received packet, *packet*; maps *counts* and *congested_counts* from tuples of (switch ID, output ports) to non-negative integers representing an estimate of the number of packets (respectively packets that encountered congestion) that passed through the specified output port. Both maps should be initialized to 0 prior to receiving any packets.

1. **if** *packet.hop_count* > 0 **then**
   
   **2**
   
   \[ \text{counts}[\text{packet.hop_reservoir}] \leftarrow \text{counts}[\text{packet.hop_reservoir}] + \text{packet.hop_count} \]

3. **if** *packet.congested_count* > 0 **then**
   
   **4**
   
   \[ \text{congested_counts}[\text{packet.congested_reservoir}] \leftarrow \text{congested_counts}[\text{packet.congested_reservoir}] + \text{packet.congested_count} \]

The **if** statement on line 6 of algorithm 3 is intentionally vague. It is difficult to give a precise definition that extends beyond a single switch design and network topology [7, Section 15.1]. There is also some flexibility in when the congestion status of the output port is measured. For the analysis in this thesis, we classify an output port as congested if, at the time a packet is added to the queue, the length of the queue exceeds its available credits. Some interesting alternative definitions are discussed in section 5.10.

Finally, note that on line 2 and line 4 of algorithm 4, the cumulative counts are incremented by the number of hops instead of by one. Thus, the recorded counts are essentially weighted counts of the raw number of packets containing a given link. This
can be thought of as charging the entire path to the recorded sample. It ensures that paths of different lengths are combined appropriately and that the values in counts and congested_counts are accurate estimators of the number of packets (respectively, congested packets) that travel through each link.

For example, if a compute node receives $N$ packets from a sender 3 hops away and $M$ packets from a sender 5 hops away, and the receiver node only has 1 link to the network, we expect $N/3 + M/5$ packets to contain the switch ID and port number of that sole link between the network and the receiver. Now suppose the second sender is only 4 hops away instead of 5 but still sends $M$ packets. We expect the number of packets containing the ID of the link to the receiver to increase to $N/3 + M/4$, in spite of the fact that the same amount of traffic flows through the link. The fact that a global change (moving the second sender) changes the raw counts, a property that seems like it should be local, suggests that raw counts are not an appropriate metric for network traffic. In this scenario, the weighted count reports an expected $N + M$ count for the link between the receiver and the network regardless of the receiver’s distance to the senders.

Although the weighted packet and congested packet counts are important for analysis, a single value that is easy to visualize (see section 3.4) is the congested_fraction, defined as the fraction of packets going through a link that were marked as congested at that link. In the same way counts[e] is an estimate of the number of packets that traveled through a link $e$, congested_counts[e] is an estimate of the number of packets that experienced congestion on link $e$. Thus, we can compute the
congested fraction as

\[
\text{congested\_fraction}[e] = \begin{cases} \frac{\text{congested\_counts}[e]}{\text{counts}[e]} & \text{counts}[e] > 0 \\ 0 & \text{counts}[e] = 0 \end{cases}.
\]

Although we expect \(\text{congested\_counts}[e] \leq \text{counts}[e]\) for every link \(e\), because these values are sampled, it is possible to have \(\text{congested\_counts}[e] > 0\) while \(\text{counts}[e] = 0\). This does occur in practice, but becomes exceedingly improbable as more packets are sampled.

This reservoir sampling-based scheme does not immediately provide a way to capture calling context or time-variant information. However, because the information is collected on compute nodes rather than on networking devices, the algorithms can be modified to support collecting both types of information. For example, prior to sending a packet, the sender can record the identifier (e.g. InfiniBand sequence number, or an MPI-level identifier), along with the calling context that caused the packet to be sent. Upon receiving the packet, the receiver can record the packet identifier, perhaps along with the calling context of the receive call. To keep the book keeping requirements low, the sender might only collect this information about a subset of the outgoing packets, which it would tag to ensure the receiver also collected calling context information.

A simple modification to support collecting time-variant information is to periodically store and reset \(\text{counts}\) and \(\text{congested\_counts}\). Since \(\text{counts}\) and \(\text{congested\_counts}\) can be stored sparsely, this can be implemented fairly efficiently.
3.3 Statistical analysis

We would like to explore the statistical properties of the output of this algorithm prior to using it to make decisions. Our primary assumption is that the random number generator used on line 2 and line 7 of algorithm 3 is ideal. Under this assumption, reservoir sampling computes a perfect uniform random sample [21].

We begin our analysis with the simplest scenario. Suppose we sample $N$ packets taking the same path of length $l$, and let $e$ be a link along the path, identified by its switch ID and output port number. Furthermore, suppose these $N$ packets are the only traffic on the network, and no congestion occurs.

Then the number of packets containing $e$ in the hop reservoir when they are received is a binomially distributed [64] random variable $R \sim \text{Binomial}(N, 1/l)$. $R$ has mean $\mathbb{E}[R] = N/l$ and variance $\text{Var}[R] = N(1/l)(1 - 1/l)$. We can then define the weighted count, i.e., the value of $\text{counts}[e]$, $W = l \times R$. The expected value of $W$ is $\mathbb{E}[W] = l \times \mathbb{E}[R] = l \times N/l = N$, as previously mentioned. The variance, $\text{Var}[W] = l^2 \text{Var}[R] = l^2 \times N(1/l)(1 - 1/l) = N(l - 1)$. This gives a coefficient of variation $\frac{\sqrt{\text{Var}[W]}}{\mathbb{E}[W]} = \frac{\sqrt{l-1}}{\sqrt{N}}$. Since $1/\sqrt{N}$ decreases fairly slowly, this means we need a large number of samples to ensure high relative accuracy (i.e., a high probability of a low percent error). For example, we need to sample more than $10000(l - 1)$ packets to have a coefficient of variation less than 1%. We expect the relative size of the weighted counts to be more useful than their exact values, so decreasing the coefficient of variation is not a primary concern.

We would like to begin relaxing some of the assumptions in this analysis. First suppose that $N_1, N_2, \ldots, N_c$ packets flow along paths of respective lengths $l_1, l_2, \ldots, l_c$.

---

*As a matter of simplicity and convention, we measure length in hops, ignoring the link between the sender and the first switch, but not ignoring the link between the last switch and the receiver.*
all containing edge $e$, and all ending at the same NIC (or that the values from the
destination NICs have been aggregated). Then, as before, we can define $R_1 \sim Binomial(N_1, 1/l_1), \ldots, R_c \sim Binomial(N_c, 1/l_c)$, and $W = \sum_{i=1}^{c} l_i R_i$. By the li-
nearity of expectation, $E[W] = \sum_{i=1}^{c} E[l_i R_i] = \sum_{i=1}^{c} l_i * \frac{N_i}{l_i} = \sum_{i=1}^{c} N_i$. Thus,
$E[W] = \sum_{i=1}^{c} N_i$, the number of packets that flow through edge $e$ in this scena-
rio, just as in the single path scenario. This proves the claims from section 3.2 that
the weighted sum, i.e. $W$, behaves well with an arbitrary number of paths.

Next, we’d like to analyze $congested\_counts[e]$. Again, restricting the traffic to a
single path will make the analysis more tractable. There are several choices for mo-
deling congestion, but a simple one is to assume that $M$ of the $N$ packets encountered
congestion on $g$ of $l$ hops, with $M \leq N, g \leq l$, and $e$ one of the $g$ hops. As before,
we’ll model the raw count of packets containing $e$ in the congested reservoir upon
arrival with $\hat{R} \sim Binomial(M, 1/g)$. Then $\hat{W} = g * \hat{R}$ models the weighted congestion
count, i.e., the value of $congested\_count[e]$.

To combine the analysis of $W$ and $\hat{W}$, first notice that they are independent as
random variables. Although certainly not independent at a conceptual level (e.g. it
is far more likely for $\hat{W} < W$ to be true than $\hat{W} > W$), this is due entirely to the
relationship between $M$ and $N$. Independence as random variables comes from the
independence between the sampling to fill $hop\_reservoir$ and $congested\_reservoir$,
which is ultimately derived from the source of random numbers. Generating a random
number on line 2 of algorithm 3 does not bias the random number generated on line 7.

Now, put $F = \hat{W}/W$, the congested fraction. Quotients of random variables are,
in general, difficult to analyze, but using a third order Taylor series-based expansion
technique [65], we can estimate
\[
E[F] \approx \frac{E[\hat{W}]}{E[W]} + \frac{E[\hat{W}] \text{Var}[W]}{E[W]^3} - \frac{E[\hat{W}] E[(W - E[W])^3]}{E[W]^4}
\]
\[
= \frac{M}{N} + \frac{M(l - 1)}{N^2} - \frac{M(l - 1)(l - 2)}{N^3}.
\]

For sufficiently large \(N\), we can assume \(E[F] \approx M/N\), as desired.

Using a similar approach, we can estimate
\[
\text{Var}[F] \approx \left( \frac{E[\hat{W}]}{E[W]} \right)^2 \left( \frac{\text{Var}[\hat{W}]}{E[\hat{W}]^2} + \frac{\text{Var}[W]}{E[W]^2} \right)
\]
\[
= \left( \frac{M^2}{N^2} \right) \left( \frac{g - 1}{M} + \frac{l - 1}{N} \right).
\]

Assuming \(M = \Theta(N)\) (i.e., \(M = O(N), N = O(M)\)), and \(g, l = O(1)\), then \(\text{Var}[F] = O(1/N)\). Since we expect \(0 \leq F \leq 1\), as opposed to \(W\) or \(\hat{W}\), which may be many thousand, it’s more important for \(\text{Var}[F]\) to be fairly small.

In both estimates, the asymmetry between the dependence on \(N\) and \(M\) is a little surprising. The fact that receiving more samples improves our estimate of \(F\) is quite intuitive, but in essence, these approximations show the stronger statement that we can estimate \(M/N\) accurately with \(F\) when \(N\) is large, even if \(M\) is not. This means that with enough packets, our reservoir sampling-based approach allows us to accurately estimate the fraction of packets that encounter congestion, even if that fraction is low.

Finally, we can also extend our analysis of the congested fraction to multiple paths. As before, we have \(N_1, N_2, \ldots, N_c\) flows along paths of respective lengths \(l_1, l_2, \ldots, l_c\). \(M_i\) packets from flow \(i\) encounter congestion on a subset of size \(g_i\) of the hops along the path including link \(e\). This implies that for each \(i\), \(0 \leq M_i \leq N_i\), and \(0 < g_i \leq l_i\). With \(R_1, R_2, \ldots, R_c\) and \(\hat{R}_1, \hat{R}_2, \ldots, \hat{R}_c\) defined as before, we put \(W = \sum_{i=1}^c l_i R_i\) and \(\hat{W} = \sum_{i=1}^c g_i \hat{R}_i\). Define \(N = \sum_{i=1}^c N_i\), and \(M = \sum_{i=1}^c M_i\). Since \(R_i\) and \(\hat{R}_i\) are all
independent, $E[W] = \sum_{i=1}^{c} N_i = N$ and $E[\hat{W}] = \sum_{i=1}^{c} M_i = M$. Using Taylor series to expand $F = \hat{W}/W$, 

$$E[F] \approx \frac{M}{N} + \frac{M \left( \sum_{i=1}^{c} N_i(l_i - 1) \right)}{N^3} - \frac{M \left( \sum_{i=1}^{c} N_i(l_i - 1)(l_i - 2) \right)}{N^4}.$$ 

Note that this approximation relies on the fact that the equivalent of the Bienaymé formula holds for the third central moment, i.e. $E[\left((X + Y) - E[X + Y]\right)^3] = E[(X - E[X])^3] + E[(Y - E[Y])^3]$. Just as before, for sufficiently large values of $N$, $E[F] \approx M/N$.

Additionally,

$$\text{Var}[F] \approx \left(\frac{M^2}{N^2}\right) \left(\frac{\sum_{i=1}^{c} M_i(g_i - 1)}{M^2} + \frac{\sum_{i=1}^{c} N_i(l_i - 1)}{N^2}\right),$$

which also decreases like $1/N$. Thus the main statistical properties of $F$ remain the same, even when considering a large number of paths.

### 3.4 Visualization methodology

A powerful tool for visualizing congestion on a network is a congested fraction plot, for example the 64-node fat tree in Figure 3.1.

Switches are indicated by light blue circles (○), and compute nodes are shown on the bottom of the plot as rectangles with thin, light blue outlines (■). One property of Clos networks, of which the fat tree topology is a sub-type, is that each switch is part of exactly one stage, and each switch only connects to switches in the previous stage and subsequent stage [66]. In congested fraction plots, each stage is arranged horizontally. Since we primarily model fat trees with three levels, we refer to the switches in the bottom level as *leaf switches*, in the middle level as *aggregation switches*, and in the top level as *core switches*. 
Figure 3.1: A sample congested fraction plot.

Links between switches are shaded based on their congestion. Links with a low congested fraction are thin and light gray, while heavily congested links are thick and dark. Links with no congestion may not appear at all. To save space, the links between leaf switches and compute nodes are omitted, and the compute node itself is shaded with the color of the link. The compute nodes arranged in a column are all connected to the leaf switch just above the column. Compute nodes are numbered top to bottom, left to right. In Figure 3.1, a few nodes are labeled with their node number.

The links in the depicted networks are full duplex, which means a link can become congested in both directions independently. Typically, the shading in congested fraction plots is the average of the congestion in both directions. In figures where the link direction is important, two diagrams will be shown, one depicting congestion
in the “down” direction of the links (i.e., towards the compute nodes) and another diagram for the “up” direction of the links (i.e., towards the core switches).

Since congested fraction plots are extremely wide for many networks, they may be cropped. Unless otherwise noted, the left-most portion of the network of is shown in the cropped plot.

### 3.5 Simulation methodology

Since algorithm 3, the core of the idea, must run on switches that are not programmable, all evaluation comes from simulated experiments.

We modified the fat tree network model in CODES to run algorithm 3, while a post-processing step computes the final counts, effectively by running algorithm 4. Although most experiments use synthetic traffic patterns, we also use some MPI trace-based experiments using TraceR [46].

Most experiments use one of two networks, roughly based on the network of the future Summit system at ORNL [67]. The two networks are detailed in Table 3.1. Both networks use 100 Gbps (e.g., EDR) technology and employ adaptive routing.

*Full* has full bisection bandwidth while *Tapered* has half bisection bandwidth. *Tapered* is tapered at a 2:1 ratio between the leaf switches and the aggregation switches, which means that each 48-port leaf switch has 32 links to compute nodes and only 16 links to aggregation switches. Cost is a major driver of tapering, since in this case, it reduces \( \frac{\text{number of links}}{\text{number of compute nodes}} \) from 5 to 3, a significant savings. Additionally, *Tapered* has far fewer expensive optical links than *Full*, increasing the cost difference.
<table>
<thead>
<tr>
<th>Name</th>
<th>Nodes</th>
<th>Taper</th>
<th>Link</th>
<th>Rails</th>
<th>Planes</th>
<th>Nodes per leaf</th>
<th>Switches</th>
<th>Switches</th>
<th>Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Band-</td>
<td></td>
<td></td>
<td>width</td>
<td></td>
<td></td>
<td>Radix</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>width</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full</td>
<td>3564</td>
<td>No taper</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>18</td>
<td>504</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Tapered</td>
<td>4608</td>
<td>2:1 taper</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>32</td>
<td>288</td>
<td>48</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: A comparison of the simulated networks used in this thesis
Chapter 4

Case studies and applications

4.1 Gaining insight into traffic and congestion

This thesis focuses on understanding three specific causes of congestion:

- a poor communication pattern,
- a poor mapping between an application’s logical topology and the physical network topology, and
- background traffic due to another user’s application.

Although these three causes are not the root cause of every case of congestion*, distinguishing between these three causes is useful because each cause requires a different approach to fix.

A poor communication pattern is inherently impossible for the network to service without causing congestion. Fixing a poor communication pattern requires changing the application code and is the most difficult to fix. Possible fixes include switching to a communication-avoiding algorithm [69] and using well-implemented MPI collectives [3, 70] instead of a large number of point-to-point messages.

---

*For example, an application might communicate in an intense, congestion-causing communication phase instead of spreading the communication out and overlapping it with computation [68]. The techniques discussed in this chapter might improve the performance of the application slightly, but will not address the root cause.
A poor mapping between an application’s logical and physical topologies can be fixed by changing the mapping. The mapping is normally chosen by the batch scheduler at runtime, but can also be manually specified. Since the mapping is normally transparent to the application, changing it often does not require source code changes, or even recompilation. Although applying a better mapping is easy, determining which mapping to apply may be difficult.

When an application experiences congestion due to traffic from another user’s application, there is often little that can be done other than waiting. The most foolproof way to determine the performance impact due to extraneous traffic is to run the application in an environment without extraneous traffic. Unfortunately, this is rarely practical, and reservoir sampling gives an alternative method to determine when this is the problem. Tuning the congestion control protocol parameters (see section 2.2) may also lessen the impact of one application on another.

Our methods for distinguishing between the three causes use the notion of a traffic tree, an extension of the idea of a congestion tree. A traffic tree is a union of paths along the network topology, with switches as vertices and links as directed edges. A traffic tree is typically a tree, but may be a directed acyclic graph with multiple roots. When one link on a traffic tree (including a virtual link between the NIC and the application) becomes oversubscribed, the destination of the link becomes the root of a congestion tree. Queues in the source of the link begin to fill up, reducing the effective bandwidth of the links along paths that pass through the initially oversubscribed link, often causing them to congest in addition. If the traffic causing oversubscription of a link persists, the congestion spreads in a tree-like pattern. Given enough time, if unaddressed, all links along a path containing the the root of the congestion tree become congested.
**Definition 4.1.** *Endpoint congestion* is congestion where the root of the congestion tree is a compute node.

**Theorem 4.1.** On a network where each compute node runs one logical task and has equal injection and ejection bandwidth, endpoint congestion cannot be eliminated by a change in the mapping between the logical and physical network topologies.

*Proof.* If the root of the congestion tree is a compute node $r$, then, by definition, the link leading to $r$ is oversubscribed. This means that the sum of the desired injected bandwidth of other nodes destined for $r$ exceeds the ejection bandwidth of $r$. Let $M$ be the current mapping from the logical network topology to the physical network topology. Then there is some logical node $l$ such that $M(l) = r$. Suppose $M'$ is another mapping, and let $r' = M'(l)$. Since the traffic destined for $r$ in the original mapping is really traffic destined for $l$ at a logical level, under the mapping $M'$, that traffic will be destined for $r'$. Since each node in the network has equal ejection bandwidth, in particular, $r'$ has the same ejection bandwidth as $r$, which is less than the desired bandwidth of the messages injected by other nodes for $r'$. Thus the link leading to $r'$ would be oversubscribed under the mapping $M'$, causing another congestion tree. So, changing the mapping between the logical and physical topologies cannot eliminated endpoint congestion. \qed

From the proper perspective, Theorem 4.1 also applies in the case of multiple MPI ranks per physical compute node. There are multiple logical nodes per physical node in this situation, and the network is extended to include on-node communication paths. This situation is more difficult to model accurately, so it will normally not be considered explicitly.

Note that Theorem 4.1 does not imply that a different mapping cannot lessen the
overall performance impact of endpoint congestion, only that it cannot be eliminated.

**Theorem 4.2.** Suppose that a traffic tree contains a congestion tree with root $r$. Then the bandwidth of the oversubscribed edges in the traffic tree is fully utilized.

**Proof.** Let $r$ be the root of a congestion tree, and let $e$ be an oversubscribed link into $r$. Since $r$ is the root, any outgoing edges in the traffic tree are not oversubscribed. This means that $r$ has the capacity to forward the full capacity of $e$. Since $e$ is oversubscribed, the source of $e$ has enough traffic to saturate $e$, so it will utilize the maximum amount of bandwidth of $e$ as possible, which fully utilizes the link.

Theorem 4.2 is particularly powerful because it allows us to infer information about the global status of the network from reservoir sampling, that observes traffic from only one application. If we observe with reservoir sampling a congestion tree with heavily congested links to the root of the congestion tree, and our sampling measurements of network traffic indicate that those links have substantially less outgoing traffic than the link bandwidth, then we know there is other traffic we are not observing, i.e. traffic from another application.

### 4.2 Suggesting a better mapping

We begin with a simple synthetic 2-D stencil communication pattern on an entire system with a tapered fat tree network topology (Tapered in Table 3.1). The logical topology is a $64 \times 72$ two-dimensional grid. Each logical compute node is responsible for a portion of the mesh, and at the end of each iteration, exchanges boundary values with each of its logical neighbors in sequence. Nodes on the edge of the logical topology simply wait instead of sending to a neighbor outside of the problem domain. As a synthetic pattern, this models the equivalent of 1 MPI rank per node, i.e. the
For simplicity, we simulate one round of communication and no computation.

The simplest way to map the logical topology to the physical topology of compute nodes numbered 0, 1, ..., 4607 is a row major mapping: (0, 0) → 0; (1, 0) → 1; (2, 0) → 2; ... (63, 0) → 63; (0, 1) → 64; ... This mapping causes a large amount of congestion, as shown in Figure 4.1.

Closer examination shows that the congestion is most severe during the +y phase, i.e., when (0, 0) sends to (0, 1). Figure 4.2 shows the congestion during this phase. Notice that even though overall congestion is severe, congestion at the endpoints is minimal. This indicates that the bottleneck is internal to the network, and the congestion is not due to oversubscription of an endpoint. Since every compute node is connected to a different leaf switch than its +y neighbor, routing this pattern without congestion requires that the bandwidth between the leaf switches and aggregation switches be at least as large as the aggregate injection bandwidth of the compute nodes running the experiment. Since the simulated network has a 2:1 taper at this
(a) Congestion on the down-facing links during the $+y$ phase.

(b) Congestion on the up-facing links during the $+y$ phase.

Figure 4.2: Congested fraction plots during the $+y$ phase of the stencil communication pattern with the row major mapping.
level, this pattern oversubscribes the bandwidth at the second level, resulting in congestion.

Although not immediately apparent, Figure 4.2 demonstrates the exact behavior the theory predicts. In particular, the lack of congestion in Figure 4.2a is because the root of the congestion trees are at the aggregation switches. This can be seen more clearly in Figure 4.3, which only shows traffic arriving at compute node 1446, an arbitrary, fairly representative node. The path annotated with arrows shows one (fairly degenerate) congestion tree, consisting only of the link labeled B. † Returning to the definition of congestion, this means that the leaf switch at the origin of B (leaf switch 43) is unable to send as many packets as it would like to the aggregation switch at the destination of B (aggregation switch 175). During this phase, all 32 compute

---

†The link labeled A should be part of the congestion tree, but because our reservoir sampling scheme only observes congestion as a packet leaves a switch, not a compute node, it is blind to congestion on link A.
nodes connected to leaf switch 43 send a message to their +y neighbors connected to the same leaf switch as node 1446. Leaf switch 43 has 200 Gbps of incoming traffic destined for aggregation switch 175, but only a 100 Gbps link, so the output queue for that link saturates. Since the only traffic for the link labeled C comes from link B, link C cannot become congested unless the leaf switch at the end of link C cannot accept traffic at 100 Gbps for any reason, which does not occur in this experiment.

Since reservoir sampling provides an estimate of the number of packets that flow though a given link, with the packet size and a rough idea of time, we can actually estimate the utilized bandwidth of link C. Using packet arrival time as an estimate for the time that the packet crossed link C, we calculate an estimated bandwidth of 101.9 Gbps, which suggests it is fully utilized as predicted.

Thus, according to Theorem 4.1 and Theorem 4.2, the congestion in Figure 4.1 is due to a poor mapping between the logical and physical topologies. In general, determining a better mapping may be a difficult problem. We explore two general automatic methods, and then provide a problem-specific expert-based mapping.

4.2.1 Reverse Cuthill-McKee

The reverse Cuthill-McKee algorithm [71, Chapter 4.5] reorders the rows and columns of a sparse matrix to minimize the width of bands of non-zero elements by moving non-zero elements closer to the diagonal of the matrix. To attempt to improve communication performance, we apply the algorithm to the adjacency matrix of the communication graph, i.e. the binary matrix where the (i, j) element is 1 if and only if node i and node j communicate during an application. If all non-zero elements of the adjacency matrix are near the diagonal, then |i − j| is small for any pair of nodes i and j that communicate. Since i and j are connected to the same leaf switch
if and only if \( \lfloor i/32 \rfloor = \lfloor j/32 \rfloor \), keeping \(|i - j|\) small may improve the communication locality. Certainly, if \(|i - j| \geq 32\), then \( \lfloor i/32 \rfloor \neq \lfloor j/32 \rfloor \). The communication adjacency matrix before and after running reverse Cuthill-McKee is shown in Figure 4.4.

One significant weakness of Cuthill-McKee algorithms is that they do not consider the volume of traffic sent between a pair of nodes that communicate. All messages in this stencil pattern are the same size, but that may not be true for other applications.

Figure 4.4: The adjacency matrix of the communication graph for the 2-D stencil pattern with the row major mapping (black) and the reverse Cuthill-McKee mapping (red).
Figure 4.5: Congested fraction plot of the 2-D stencil pattern with a reverse Cuthill-McKee mapping.

Unfortunately, when applied to the stencil mapping on the tapered fat tree, the reverse Cuthill-McKee mapping performs poorly. Congestion with the reverse Cuthill-McKee mapping is as bad as with the row major mapping, as shown in Figure 4.5. Notice how the more narrow band at the lower and higher rows in Figure 4.4 corresponds to a reduction in congestion at low and high number compute nodes in Figure 4.5.

Although the algorithm succeeds in decreasing $|i - j|$ for many communication partners, it mostly loses the locality that the row major mapping has in one dimension. The median value of $|i - j|$ decreases from around 32 to around 17, but the fraction of communication pairs with $|i - j| \geq 32$ increases from 0.5 to 0.79. Thus the reverse Cuthill-McKee mapping increases the necessary bandwidth between leaf switches, worsening congestion and decreasing performance.
4.2.2 Graph partitioning

Another automatic approach to improve the mapping between logical and physical topologies is to use graph partitioning. Given an undirected graph with weighted edges, graph partitioning algorithms attempt to find an assignment of vertices to approximately equal-sized partitions that minimizes the total weight of edges with endpoints in different partitions. If the vertices in the graph correspond to compute nodes, edge weights correspond to the volume of communication between compute nodes, and partitions correspond to leaf switches, then a good graph partitioning corresponds to a mapping with low aggregation switch-level bandwidth requirements. Unfortunately, common graph partitioning algorithms typically do not guarantee exactly equal-sized partitions, even in the case where the number of vertices is divisible by the number of partitions. Since the number of compute nodes attached to a leaf switch is fixed at 32 for this network, partitionings in which any partition contains more than 32 vertices do not translate to valid mappings.

We used Metis [72] in multilevel recursive bisectioning mode to assign each logical compute node to a leaf switch. Since Metis allocated one extra node to 14 partitions, in the post-processing phase that creates a mapping based on the partitioning, a few nodes were moved to other partitions. This is a fairly low fraction of the nodes, so we expect the impact of moving these 14 nodes to be negligible.

We also explored an incremental graph partitioning algorithm. An incremental algorithm could modify a mapping used for a previous execution of the application, taking into account congestion measurements during the execution. We implemented a variant of the Kernighan-Lin algorithm [73] that works with an arbitrary number of partitions.

For each partition $p_1$, we choose a partition $p_2$ with the most edges from $p_1$. We
Figure 4.6: Congested fraction plot of the stencil communication pattern using a mapping produced with the Metis partitioner.

Figure 4.7: Congested fraction plot of the stencil communication pattern using a mapping derived from our incremental graph partitioning algorithm.
then swap one node from $p_1$ with one node from $p_2$ using the Kernighan-Lin heuristic. Since this algorithm only swaps individual nodes, it preserves the number of vertices in each partition. If the algorithm starts with a valid partitioning, the resulting partitioning must also be valid.

Both graph partitioning algorithms produced mappings that increased locality and dramatically reduced congestion. Metis’s partitioning had an edge cut fraction of 0.19 (i.e., 81% of edges are between two vertices in the same partition), while our incremental algorithm’s partitioning had an edge cut fraction of 0.23. With these smaller edge cuts, the bandwidth between leaf switches and aggregation switches was no longer oversubscribed, so no major congestion trees formed. The mapping based on Metis’s partitioning eliminated all congestion. With the mapping based on incremental graph partitioning, almost all congestion was eliminated; only approximately 0.05% of all hops were congested.

4.2.3 Geometric approach

Given the simplicity of the communication pattern, geometric techniques can provide a provably optimal pattern for minimizing traffic at the second level switches. Although as discussed in section 2.4, this may not be optimal for overall performance, Figure 4.2 strongly suggests that it will help. Consider the grid formed by compute nodes placed in the logical topology. If we form the shape $S_i$ consisting of the regions of the grid belonging to nodes connected to leaf switch $i$, then the total traffic that leaves leaf switch $i$ is proportional to the perimeter of $S_i$, excluding the boundary of the domain. Since the number of compute nodes connected to each leaf switch is fixed, the area of $S_i$ is fixed. We use $P(S_i)$ to denote perimeter and $A(S_i)$ to denote area for the shape $S_i$. 
The problem of minimizing the traffic leaving leaf switch $i$ is equivalent to finding the shape of minimum perimeter for a given area, which is well known to be a circle. Unfortunately, we need the additional constraint that $S_i$ is tileable with unit squares for this equivalence to provide a feasible communication-reducing mapping, but a circle provides a lower bound that $P(S_i) \geq 2\sqrt{\pi \sqrt{A(S_i)}} \approx 3.54\sqrt{A(S_i)}$. This means that for any possible mapping of the stencil communication pattern, at least $3.54\sqrt{A(S_i)}$ messages must leave any leaf switch containing $A(S_i)$ nodes mapped to the interior of the logical topology and no nodes mapped to the boundary. A small corrective factor must be applied to this bound for nodes along the boundary, since they have less than four communication partners.

Any shape $S$ of area $A(S_i)$ with only angles of measure $90k^\circ$ for $k \in \mathbb{Z}$ has a perimeter at least as large as a rectangle of area $A(S_i)$. \footnote{To see this, notice that if $S$ contains two edges that form an angle of $-90^\circ$, these edges can be flipped in a shape $S'$ to form an angle of $90^\circ$, which increases the area and not changing the perimeter. This flipping operation increases the area of $S'$ while not changing the perimeter, so $S'$ can be scaled down to $S''$ with the same area as $S$ and a strictly smaller perimeter. In certain cases, this may cause a $180^\circ$ angle to be formed, in which case the two line segments can be deleted in a shape $S'$. $S'$ has the same area as $S$ but a smaller perimeter.}

Thus, since any shape that is tileable with unit squares only has angles of measure $90k^\circ$ for $k \in \mathbb{Z}$, this provides a more realistic lower bound. It’s also well known that the rectangle with the lowest perimeter for a given area is a square, so $P(S_i) \geq 4\sqrt{A(S_i)}$.

However, if $\sqrt{A(S_i)}$ is not an integer, this bound may not be achievable. The naïve approach is simply to find the integers closest to $\sqrt{A(S_i)}$ that divide $A(S_i)$, likely by computing the prime factorization of $A(S_i)$. In practice, this seems to work well, but it is not always optimal, especially when $A(S_i)$ does not have any factors close to $\sqrt{A(S_i)}$. For example, when $A(S_i) = 5$, the bounds guarantee that $P(S_i) \geq 8.9$, and
a $5 \times 1$ rectangle has perimeter 12, but there is in fact a unit-tileable shape (i.e., a shape that can be tiled with unit squares) with area 5 and perimeter 10 (Figure 4.8).

In this case, where $A(S_i) = 32$, an $8 \times 4$ rectangle works fairly well, since it’s easy to tile the $64 \times 72$ domain with $8 \times 4$ rectangles. This tiling converts to a mapping where the compute nodes responsible for an $8 \times 4$ portion of the domain are connected to a single leaf switch. The geometry does not provide an exact mapping of which nodes should be responsible for which portion inside the $8 \times 4$ sub-domain, or which $8 \times 4$ sub-domain should be allocated to which leaf switch. However, if our goal is just to minimize the traffic from one leaf switch to another, as the congested fraction plot suggested, the exact mapping does not matter.

Simulating the same stencil communication pattern with a mapping that assigns one $8 \times 4$ rectangle of the logical topology to each leaf switch yields a significant improvement in wallclock time and a complete elimination of congestion. The congested fraction plot is shown in Figure 4.9. This technique could be extended to tile at multiple levels, but given that tiling at a single level was enough to eliminate all congestion, we do not pursue it further.
4.2.4 Discussion of mapping improvements

With reservoir sampling, we were able to diagnose that the congestion with the 2-D stencil pattern was due to a bad mapping, and in particular due to a lack of locality at the leaf switches. Using this diagnosis, we chose techniques to improve locality at the leaf switches, and we observed a significant increase in performance and a reduction of congestion.

An overall comparison of the performance of the various techniques for improving the 2-D stencil pattern is shown in Table 4.1. For comparison, the table includes a mapping that is a random permutation of the logical topology.

The geometry-based tiling mapping and the ordinary graph partitioning mapping performed very well, effectively tying for the best mapping. These two eliminated all congestion and demonstrated significant speedup. The mapping based on incremental graph partitioning also increased locality, dramatically reduced congestion, and performed very well.

Without the diagnosis from reservoir sampling, we may have tried topology-
Figure 4.10: Congested fraction plot of the stencil communication pattern using a random mapping.

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Elapsed time for one round</th>
<th>Average path length</th>
<th>Congested fraction plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>86054</td>
<td>4.657</td>
<td>Figure 4.10</td>
</tr>
<tr>
<td>Row major</td>
<td>61186</td>
<td>2.087</td>
<td>Figure 4.1</td>
</tr>
<tr>
<td>Reverse</td>
<td>89020</td>
<td>2.967</td>
<td>Figure 4.5</td>
</tr>
<tr>
<td>Cuthill-McKee</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ordinary graph</td>
<td>44018</td>
<td>1.474</td>
<td>Figure 4.6</td>
</tr>
<tr>
<td>partitioning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incremental graph</td>
<td>48925</td>
<td>1.623</td>
<td>Figure 4.7</td>
</tr>
<tr>
<td>partitioning</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One level tiling</td>
<td>44035</td>
<td>1.433</td>
<td>Figure 4.9</td>
</tr>
<tr>
<td>Two level tiling</td>
<td>–</td>
<td>1.374</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 4.1: A comparison of the different mappings used with the 2-D stencil pattern.
oblivious techniques, like adding randomness to the mapping, or using reverse Cuthill-McKee. Adding randomness might have lessened the severity of a single hotspot, but it exhibits much worse locality than the row major mapping. Reverse Cuthill-McKee did improve the locality according to one metric, the median value of the rank-distance between communication partners, but because it does not consider the network topology, it could not specifically target the problem of insufficient locality at the leaf switches. Both topology-oblivious techniques actually increased congestion and decreased performance.

4.3 Fundamental congestion caused by a communication pattern

Here, we consider a reduction communication pattern: suppose 1024 nodes want to aggregate a final computation result consisting of 50 arrays of 1024 integers (4 KB) each. These experiments are trace-driven from skeleton MPI applications using TraceR, as discussed in subsection 2.5.1, and run on the full bisection bandwidth fat tree model described in Table 3.1.

The naïve approach is for each MPI rank to send its result to a fixed rank, e.g., rank 0, which can then compute the reduction locally. This pattern is similar to the hotspot pattern discussed in section 2.2, and because traffic to the root of the reduction greatly exceeds its ejection bandwidth, we expect this pattern to cause endpoint congestion. Figure 4.11 shows the spreading of the congestion tree, as predicted by Pfister and Norton [23]. Here, we show that our reservoir sampling-based technique enables us to capture time-varying behavior with much higher fidelity than, for example, Mellanox’s switch counter-based technique, which samples at a maximum frequency
of 2 seconds [40]. Each of the five congested fraction plots in Figure 4.11 represents approximately 0.002 seconds over a total execution time of 0.018 seconds.

A more detailed view of the same experiment in Figure 4.12 shows the endpoint congestion more clearly. Recall that the black box immediately below the leftmost column of switches represents the congested fraction of the link between rank 0 and the left-most leaf switch, which is > 0.998. According to Theorem 4.1, this is a problem with the communication pattern that can’t be fixed by a different mapping.

To confirm this, we tried a different mapping which moved the root of the reduction to an arbitrarily chosen rank 388. The congested fraction plot (Figure 4.13) shows how changing the mapping moved the root of the congestion tree, but did not shrink it. Changing the mapping had essentially no impact (< 0.01%) on the time taken to complete the communication pattern.

Since there is only one application, we know that the cause of the congestion cannot be another user’s application, but to confirm, we should still check the utilized bandwidth of the root of the congestion tree as per Theorem 4.2. Reservoir sampling gives an estimate of 92.6 Gbps, which is fairly close to the line rate of 100 Gbps.

Since the congestion cannot be fixed by a different mapping, we need to switch communication patterns. In this case, the simplest fix is to switch to the MPI\_Reduce collective provided by MPI. MPI\_Reduce is typically implemented with a power-of-two-based reduction tree [74]. Although the collective does not substantially change the number of messages, it spreads traffic out, enabling parallelism. Furthermore, it drastically reduces the traffic to the root of the reduction, totally eliminating the endpoint congestion. The combination of parallelism and lack of congestion when using MPI\_Reduce instead of a naïve reduction pattern improved the performance of the operation by a factor of 86.9.
Figure 4.11: Chronological progression of congested fraction plots of five sub-intervals of the execution of the reduction-like pattern to rank 0. Time progresses from top to bottom and is labeled on the left of each image in milliseconds based on packet arrival time.
Figure 4.12: Congested fraction plot of the na"ive reduction pattern to rank 0.

Figure 4.13: Congested fraction plot of the na"ive reduction pattern to rank 388.
4.4 Congestion due to background traffic

For our experiments with background traffic, we use two traffic patterns running on the tapered fat tree model described in Table 3.1. The first uses a ring exchange communication pattern, where rank $i$ sends a message to rank $i - 1 \pmod{R}$, where $R$ is the number of ranks. The second traffic pattern, the background traffic pattern, is uniform random, where each rank picks a pseudo-random destination rank for each message. Both applications use a constant message size of 128 KB. We expect the first pattern by itself to perform well, since it exhibits good locality, and the fact that each compute node only ever receives from one sender guarantees endpoint congestion cannot occur. One complicating factor is that we use a node allocation that mimics the scatter of real allocations from a batch scheduler for the two applications, rather than directly interleaving the nodes or using the first or last half of the compute nodes. We allocate node $i$ to the primary application when $(i + 1)^2$ has an even number of 1 bits, and to the background traffic application when $(i + 1)^2$ has an odd number of 1 bits. This essentially arbitrary function serves solely to create an irregular-looking, fairly even pattern, and it assigns 2172 nodes to the primary application, leaving 2436 nodes to create background traffic.

To confirm our expectation that the primary application should perform well by itself, we temporarily disabled the background traffic while maintaining the same set of 2172 nodes for the primary application. As shown in Figure 4.14, the pattern does not cause any congestion. It finished in 33.6 $\mu$s.

We also ran the background traffic in isolation, which did cause congestion, as shown in Figure 4.15. The congestion is primarily endpoint congestion due to multiple nodes simultaneously choosing the same random destination. The background traffic application completed in 106.2 $\mu$s.
Figure 4.14: Congested fraction plot of the primary application (ring exchange) running in isolation on 2172 nodes distributed throughout the system.

Figure 4.15: Congested fraction plot of the background traffic (uniform random traffic) running in isolation on 2436 nodes distributed throughout the system.
Figure 4.16: Congested fraction plot from the perspective of the primary application while running simultaneously with the background traffic application.

When we run both applications simultaneously, the performance of the primary suffers significantly, taking 51.4 µs to complete, an increase of 54%. Although less important, the performance of the background traffic application suffers as well, increasing by 20% to 127.1 µs.

Suppose that we are a user of the primary application, but we do not have permission to monitor the global state of the network, and we do not know anything about the background traffic. Thus, when using reservoir sampling, we only have access to samples collected with packets sent by the primary application to other nodes running the primary application. We can see that our application executed much more slowly than expected, but we initially do not know the cause, other than perhaps that the communication in our application is slower than expected.

The congested fraction plot using only the samples from the primary application is shown in Figure 4.16. Notice what looks like a congestion tree rooted at leaf switch 37. The congestion tree is suspicious because there doesn’t appear to be any endpoint congestion there, and the network is not tapered there in that direction either. Using
Figure 4.17: Congested fraction plot from the perspective of a network administrator while the primary application runs simultaneously with the background traffic application.

The full traffic information and packet receipt time, we estimate the utilized bandwidth of the links leaving switch 37. No port exceeds 30 Gbps, substantially less than the 100 Gbps line rate. According to Theorem 4.2, this implies that there must be other traffic we cannot see that is impacting the performance of our application and is causing the congestion we detect with reservoir sampling. Although we cannot blame a specific external application, this tells us that the situation may be transient and improve next time the application is run, so we do not need to change our application. In spite of this, the application may still benefit from improvements in the communication pattern, mapping, or the tuning of the congestion control parameters.

Although a normal user would not have access to full-network information, with reservoir samples from the background traffic, we are able to understand the cause of the congestion more fully. Figure 4.17 shows the congested fraction plot with information from all nodes. Notice that in fact there is severe endpoint congestion on compute nodes connected to switch 37. Both the link between switch 37 and compute
node 1204 and the link between switch 37 and compute node 1213 (ports 20 and 29 respectively) have congested fractions above 0.9. With full traffic information, we can estimate that the utilized bandwidth of these two links is 98.5 Gbps and 88.6 Gbps respectively. Oversubscription for these two links, and, to a lesser extent some other links, causes a congestion tree to form, which causes congestion in the links from aggregation switches to switch 37. The primary application does not run on compute node 1204 or 1213, so it is not affected by congestion on links to those nodes. However, the primary application does use the links from the aggregation switches to switch 37, and when the congestion spreads to these links, it does impact the performance of the primary application.
Chapter 5

Implementation considerations and approaches

Reservoir sampling is a powerful tool for understanding network performance, but if it is restricted to applications running in network simulators, its utility is limited. To help improve real applications, support for reservoir sampling must be added to networking hardware. There is still a substantial gap between the method introduced in section 3.2 and what can be practically implemented in a next-generation switch. The goal of this chapter is to shrink the size of a few aspects of that gap.

If we design a new networking protocol, e.g., an InfiniBand replacement, or a new major version of a protocol, we have a lot of flexibility to add fields to packets. However, every byte in the header consumes some bandwidth to transmit, so we have an incentive to shrink the fields for reservoir sampling as much as possible.

However, designing a new protocol requires a major investment, and it is unlikely that any single feature justifies making such an investment. Thus, if our reservoir sampling-based technique were compatible with existing protocols, it would be much easier to implement. Most protocol headers include a small amount of unused and reserved space that future features can use without requiring major changes to the protocol specification. Our design goal is to use only this space, looking at the InfiniBand packet specification in particular, to store the reservoir in the packet. This increases the viability of our scheme for inclusion in a future version of InfiniBand or another communication protocol specification.

This chapter introduces and explores a set of modifications to the algorithms
presented in section 3.2 that dramatically reduce their space requirements. We also
discuss the requirements for and implementation of the random number generator
on the switch required by reservoir sampling. Finally, we briefly revisit the question
raised in chapter 3: when exactly is a link congested?

5.1 The current storage requirement

We should understand the storage requirement of our reservoir sampling scheme prior
to trying to reduce them. The current scheme requires storing two switch IDs, two
port numbers, and two hop counts in a packet header. Sizing the fields is tricky,
because of the trade-off between minimizing the bit count and forward compatibi-
licity. We base our sizes on the sizes of related fields found in the current InfiniBand
specification [5]:

- Switch ID (16 bits): InfiniBand assigns each switch a local ID of 16 bits. These
  IDs are only unique within a subnet, and the larger GID (global identifier)
  addresses (64-128 bits) are needed to be certain that message can ever pass
  through two switches with the same ID. Both LIDs and GIDs are used for
  switches and endpoints, i.e., compute nodes, so they are larger than necessary.
  Additionally, we do not know of any multi-subnet supercomputers at this point.

- Port number (8 bits): InfiniBand limits the port number range for a single switch
to between 1 and 254. Larger switches, e.g., the 800-port Mellanox CS8500 [75]
  are implemented as multiple logical switches in one physical container.

- Hop limit (8 bits): InfiniBand also specifies an 8-bit hop limit field in the global
  route header, but this only counts inter-subnet hops.
Using these field widths yields a total bit requirement of 64 bits. Less conservative sizes might be 10 bits for the switch ID, six bits for the port number, and four bits for the hop limit, for a total of 40 bits. For comparison, the mandatory local routing header (LRH) and base transport header (BTH) in InfiniBand are 8 and 12 bytes, respectively [5].

Adding an additional 5-8 bytes of header per packet imposes a bandwidth overhead of 0.3% – 1.5% for tagged packets. The overall impact can be reduced arbitrarily by collecting reservoir sampling information about a subset of the packets.

In an Ethernet-like environment with data-plane programmable switches [76], adding a 64 bit header is a reasonable design, and can even be done on the switch. However, adding an additional header to a switch that uses cut-through forwarding instead of store-and-forward is problematic.

One potentially viable implementation is to store the reservoir sampling fields in a footer to the packet that is not included in the CRC calculation. Still, we believe storing the reservoir sampling fields in the standard transport headers would be a simpler modification to the standard.

5.2 Probabilistic approach

We propose a probabilistic approach to replace the switch ID and port number with a single bit. The technique is based on Bond and McKinley’s information-theoretic approach for memory leak detection [77], but extended and modified to work with reservoir sampling.

Suppose we have a sequence of $Q$ records, $r_1, r_2, \cdots, r_Q$. Let $id_i$, an integer, be the ID number of $r_i$, and suppose additionally that each $r_i$ has some associated data $d_i$ taken from a finite set $D$ of possible values. For each $d \in D$, our goal is to estimate
$C_d$, the number of records in the sequence such that $d_i = d$ without storing $d_i$ in the record. Let $H : \mathbb{Z} \times D \rightarrow \{0, 1\}$ be a hash function that takes as input a record ID and a data value and returns a single bit. We discuss the formal requirements for $H$ in section 5.4, but we need for $H$ to behave like an independent Bernoulli trial [64] with probability $1/2$ for each value in $\mathbb{Z} \times D$. Suppose that instead of storing $d_i$, each record stores $id_i$ and $h_i := H(id_i, d_i)$, which occupies a single bit. For a given $d \in D$, we compute $\hat{h}_i := H(id_i, d)$ and compare it to $h_i$. For each record $r_i$, we know that if $d_i = d$, then $\hat{h}_i = h_i$. If $d_i \neq d$, then by our hypothesis on $H$, $\hat{h}_i = h_i$ is true with probability $1/2$. In other words, we expect $\hat{h}_i = h_i$ to hold for all the records for which $d_i = d$ and for half of the other records. Let $B_d$ be the number of records for which $\hat{h}_i = h_i$, and recall that $C_d$ is the number of records for which $d = d_i$. Then $B_d \approx C_d + 1/2(Q - C_d)$. Rearranging gives $C_d \approx 2 \cdot B_d - Q$, a method to estimate $C_d$, as desired.

Bond and McKinley use chunks of memory as records, the virtual address as its ID, and the calling context from which the memory was allocated as the data. We use packets as records, packet sequence numbers as IDs, and (switch ID, port number) tuples as data. With a few more modifications, this is a natural fit for our reservoir sampling scheme, which tries to estimate the counts on a per-port basis.

First, since our scheme updates the count for a (switch ID, port number) after receiving a packet, it makes more sense to consider one record $r_i$ at a time, and to compute incremental updates based on $r_i$ to $C_d$ for each $d \in D$. Receiving $r_i$ increases $N$ by 1, and also increases $B_d$ by 1 if $H(id_i, d) = H(id_i, d_i)$, so the net change to $C_d$ is $2 \cdot (+1) - 1 = +1$ if the hashes match and $2 \cdot (0) - 1 = -1$ if they do not. This also exposes an opportunity for optimization: if we know packet $r_i$ could not have traveled through $d = (\text{switch ID}, \text{port number})$ because of the source
address, we do not need to update $C_d$. This happens fairly frequently with a fat tree network, since, even with adaptive routing, the number of paths from a given source to a given destination that a packet might take is fairly limited. Other topologies may not benefit as dramatically, but even knowing simple rules, e.g., that packets does not travel through the link from a leaf switch to a compute node other than the source or destination, helps.

Secondly, we ultimately want the counts that are weighted by path length, not the raw counts. Instead of increasing or decreasing $C_d$ by 1, we increase or decrease the weighted count $W_d$ by the length of the path. The updated algorithms are shown in algorithm 5, algorithm 6, and algorithm 7.

**Algorithm 5:** Reservoir Sampling for Understanding Congestion with a Single Bit: On the sender NIC

<table>
<thead>
<tr>
<th>Input: The current outgoing packet, $packet$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $packet.hop_reservoir \leftarrow 0$</td>
</tr>
<tr>
<td>2. $packet.hop_count \leftarrow 0$</td>
</tr>
<tr>
<td>3. $packet.congested_reservoir \leftarrow 0$</td>
</tr>
<tr>
<td>4. $packet.congested_count \leftarrow 0$</td>
</tr>
</tbody>
</table>

This probabilistic scheme requires evaluating the hash function up to $|D|$ times for each packet that is received at the NIC, but it only requires switches to evaluate the hash function once per packet. With an appropriately chosen hash function, we believe this could be done within the tight timing constraints faced by the switch. It achieves a massive reduction in the number of bits needed because it replaces the switch ID and port number with a single bit.
Algorithm 6: Reservoir Sampling for Understanding Congestion with a Single Bit: On every switch

Input: The current packet, packet

1. outport ← the port through which packet will leave the current switch
2. rand1 ← a random integer between 0 and packet.hop_count, inclusive
3. if rand1 = 0 then
   4. packet.hop_bit ← H(packet.id, (my switch ID, outport))
   5. packet.hop_count ← packet.hop_count + 1
4. if outport is congested then
   7. rand2 ← a random integer between 0 and packet.congested_count, inclusive
   8. if rand2 = 0 then
      9. packet.congested_bit ← H(packet.id, (my switch ID, outport))
   10. packet.congested_count ← packet.congested_count + 1
Algorithm 7: Reservoir Sampling for Understanding Congestion with a Single Bit: On the receiver NIC

\textbf{Input}: The received packet, \textit{packet}; maps \textit{counts} and \textit{congested_counts} from the set \(D\) of tuples (switch ID, output ports) to integers representing an estimate of the number of packets (respectively packets that encountered congestion) that passed through the specified output port. Both maps should be initialized to 0 prior to receiving any packets.

1 \textbf{if} \(\text{packet.hop.count} > 0\) \textbf{then}

2 \quad \textbf{foreach} \(d \in D\) where \textit{packet} might have passed through \(d\) \textbf{do}

3 \quad \quad \textbf{if} \(H(\text{packet.id}, d) = \text{packet.hop.bit}\) \textbf{then}

4 \quad \quad \quad \textit{counts}[d] \leftarrow \textit{counts}[d] + \text{packet.hop.count}

5 \quad \quad \else

6 \quad \quad \quad \textit{counts}[d] \leftarrow \textit{counts}[d] - \text{packet.hop.count}

7 \textbf{if} \(\text{packet.congested.count} > 0\) \textbf{then}

8 \quad \textbf{foreach} \(d \in D\) where \textit{packet} might have passed through \(d\) \textbf{do}

9 \quad \quad \textbf{if} \(H(\text{packet.id}, d) = \text{packet.congested.bit}\) \textbf{then}

10 \quad \quad \quad \textit{congested_counts}[d] \leftarrow \textit{congested_counts}[d] + \text{packet.congested.count}

11 \quad \quad \else

12 \quad \quad \quad \textit{congested_counts}[d] \leftarrow \textit{congested_counts}[d] - \text{packet.congested.count}
5.3 Error analysis

Introducing this extra source of randomness changes our error estimates. Again, we start with the simplest situation: \( N \) packets taking the same path of length \( l \) containing a link \( d \). Let \( Q \) be the total number of packets that might have passed through \( d \) as in line 2 of algorithm 7. As before, we have \( W_d = 2lB_d - lQ \), where \( W_d \) is a random variable representing our estimate of \( N \), the number of packets that passed through \( d \), and \( B_d \) is a random variable representing the number of packets for which \( H(id_i, d_i) = H(id_i, d) \). Of course, as we saw before, these packets are composed of true matches, where \( d_i = d \) and false matches, where \( d_i \neq d \). Let \( R_d \) be the number of true matches, and let \( \epsilon_d \) be the number of false matches, so \( B_d = R_d + \epsilon_d \). Since \( R_d \) is just the number of times \( d \) was selected by reservoir sampling, we know that \( R_d \sim \text{Binomial}(N, 1/l) \). By our hypothesis on the properties of \( H \), \( \epsilon_d \sim \text{Binomial}(Q - R_d, 1/2) \). Notice that \( R_d \) and \( \epsilon_d \) are not independent, since the distribution for \( \epsilon_d \) depends on \( R_d \).

To calculate \( E[W_d] \), we need to calculate \( E[B_d] \) from the definition of expectation:

\[
E[B_d] = \sum_{j=0}^{Q} \sum_{k=0}^{Q-j} (j + k) \cdot P[R_d = j] \cdot P[\epsilon_d = k]
\]

\[
= \sum_{j=0}^{Q} \sum_{k=0}^{Q-j} (j + k) \cdot \left( \binom{N}{j} \left( \frac{1}{l} \right)^j \left( 1 - \frac{1}{l} \right)^{N-j} \right) \cdot \left( \binom{Q-j}{k} \left( \frac{1}{2} \right)^k \left( 1/2 \right)^{Q-j-k} \right)
\]

\[
= \frac{N}{2l} + \frac{Q}{2}.
\]

Then \( E[W_d] = 2lE[B_d] - lQ = N \) as desired.

To compute \( \text{Var}[W_d] \), we need to compute \( E[B_d^2] \). This is tractable to compute
by hand, but messy, so the computation is omitted.

$$\text{Var}[B_d] = \text{E}[B_d^2] - \text{E}[B_d]^2$$

$$= \frac{l^2Q(Q + 1) + N(2lQ - 1) + N^2}{4l^2} - \left(\frac{N}{2l} + \frac{Q}{2}\right)^2$$

$$= \frac{1}{4} \left(Q - \frac{N}{l^2}\right),$$

so $$\text{Var}[W_d] = 4l^2 \text{Var}[B_d] = Ql^2 - N.$$ It’s interesting that this is quite different from $$N(l - 1),$$ the variance of the non-probabilistic method. We will assume a that $$N$$ is a fixed fraction of $$Q,$$ i.e., $$Q = \lambda N$$ for $$\lambda \geq 1.$$ This means that if the number of total tagged packets transmitted doubles, then the number of packets through $$d$$ also doubles. With $$Q = \lambda N,$$ the coefficient of variation $$\frac{\sqrt{\text{Var}[W_d]}}{\text{E}[W_d]} = \sqrt{\lambda l^2 - 1} \ast 1/\sqrt{N}.$$ We need $$\frac{N^2-1}{l-1}$$ times as many packets as with normal reservoir sampling when using the probabilistic method.

We briefly extend the analysis to a combination of flows of multiple lengths, $$1 \leq l \leq L.$$ Let $$N_l$$ be the number of packets that pass through link $$d$$ along a path of length $$l,$$ and let $$Q_l$$ be the number of packets that traveled along a path of length $$l$$ and that, based on the packet’s source address, may have traveled through $$d.$$ Then we define $$R_{d,l} \sim \text{Binomial}(N_l, 1/l),$$ $$\epsilon_{d,l} \sim \text{Binomial}(Q_l - R_{d,l}, 1/2),$$ $$B_{d,l} = R_{d,l} + \epsilon_{d,l}.$$ Our goal is still to estimate $$\sum_{l=1}^{L} N_l,$$ which we do with $$W_d = \sum_{l=1}^{L} 2lB_{d,l} - lQ_l.$$ 

$$\text{E}[W_d] = \sum_{l=1}^{L} 2l \text{E}[B_{d,l}] - lQ_l = \sum_{l=1}^{L} N_l,$$ as desired. Also using linearity, $$\text{Var}[W_d] = \sum_{l=1}^{L} 4l^2 \text{Var}[B_{d,l}] = \sum_{l=1}^{L} Q_l l^2 - N_l.$$

Looking at one link in isolation hides a real challenge with this technique. To decrease the coefficient of variation of $$W_d,$$ for a single $$d,$$ we need to receive more packets that flow through link $$d.$$ However, receiving more packets for $$d$$ actually increases the coefficient of variation for every other link through which those extra packets may have traveled. If $$\sum_{l=1}^{L} Q_l l^2$$ is large, but $$\sum_{l=1}^{L} N_l$$ is small, i.e., we receive a large
number of packets, but not many passed through link $d$, we may spuriously observe a moderately large value of $W_d$. It might appear that we observe traffic that is impossible for a communication pattern. Bond and McKinley notice the analogous issue, and they suggest using a relatively high false positive threshold to avoid reporting erroneous memory leaks. Their strategy, when applied here, is to replace $W_d$ with 0 if we are not at least 99% sure that it did not come from a link through which 0 packets actually flowed. This happens approximately when $W_d \geq l\sqrt{Q} \Phi^{-1}(0.99)$, where $\Phi^{-1}$ is the inverse CDF of the standard normal distribution [65], and $\Phi^{-1}(0.99) \approx 2.33$.

In practice, since we don’t maintain counts for each value of $l$, we might require $W_d \geq L\sqrt{Q} \Phi^{-1}(0.99)$, where $L$ is the diameter of the network.

As an example, consider the case where one node sends to another node along a 5-hop path on the tapered fat tree. The adaptive routing algorithm can choose between 1 link to the first hop, 8 links to the second hop, 24 links to the third hop, 8 links to the fourth hop, and 1 link to the fifth hop. This gives a total of over 1500 possible paths using 400 links. Suppose the sender sends $S$ packets, and the receiver does not receive any other traffic, so $Q = S$. We want to find a lower bound on $S$ so that for each of the 400 links, $\mathbf{E}[W_d] \geq l\sqrt{Q} \Phi^{-1}(0.99)$, which means that we expect all of the links not to be flushed to zero. Under uniform adaptive routing, $S/(8 \times 24)$ packets pass through each of the links between the second and third hop. If $N_d$ is the number of packets that pass through link $d$, then this means $S \geq 8 \times 24 N_d$. Recall that $\mathbf{E}[W_d] = N_d$, so $N_d \geq l\sqrt{Q} \Phi^{-1}(0.99)$. Plugging in $l = 5$ and $Q = S$, and combining these two inequalities gives $S \geq 5 \times 10^6$ packets. Although $5 \times 10^6$ seems immense, especially given that this is not a worst case scenario, it is not unreasonable. Intel Omni-Path NICs are able to send $160 \times 10^6$ packets per second [6], and bandwidth-bound applications running on a 100 Gbps network send $3 \times 10^6$ to $6 \times 10^6$ packets per
second, depending on the packet size. For comparison, to expect to see a count of at least 1 on all 400 links without the probabilistic approach would require around 1000 packets, assuming routes are uniformly distributed among all feasible paths.

5.4 Hash functions

Given that the hash function may need to be evaluated around $10^7$ times per second, choosing it is an important design decision. The primary requirement that $H$ must satisfy is that for any collection of $Q$ distinct integers $id_1, id_2, \cdots id_Q$, any $d \in D$, and any $Q$ (not necessarily distinct) values $d_1, d_2, \cdots, d_Q \in D \setminus \{d\}$, the number $\epsilon_d$ of times that $H(id_i, d_i) = H(id_i, d)$ follows a binomial distribution with $Q$ trials and a probability of success of $1/2$. Since we can look at each $i$ as a Bernoulli trial making up the binomial distribution, this implies that for each $i$, and a $d'$ chosen uniformly from $D \setminus \{d\}$, $P[H(id_i, d') = H(id_i, d)] = 1/2$. Our choice of $d$ is arbitrary, so this is the same as $P[H(id_i, d') = 0] = 1/2$ (or equivalently, $P[H(id_i, d') = 1] = 1/2$).

For a sum of Bernoulli trials to follow a binomial distribution, the trials must also be independent. This means that for each pair of distinct $i, j$, $P[H(id_i, d') = H(id_j, d')] = 1/2$.

Bond and McKinley attempted to use $H_{parity}(id, d) = parity(id \& d)$ ($\&$ 1 in C), and found that it caused lots of false positives. This is because $H_{parity}$ violates both requirements. Since $parity(id \& d) = parity(id) \& parity(d)$, which is only 1 when both $id$ and $d$ are odd, one must make unusual assumptions about the distribution of $id$ and $d$ for $P[H(id, d) = 0] = 1/2$ to be true. They claim $H_{parity}$ is unbiased, so it is possible that they intended to use a bitwise XOR (confusing the $\&$ symbol for the $^\wedge$ operator in C) instead of AND. This would be unbiased, but it would still violate the independence requirement. Since $parity(id + 2) = parity(id)$, with
Figure 5.1: The values of $H(id, d)$ for the specified hash functions, with a fixed randomly chosen $id$, and $d$ ranging from 1 to 900, numbered left to right, top to bottom.

XOR or AND we have that $H(id_i, d) = H(id_i + 2, d)$. If $id_i$ and $id_j$ have the same parity, then $P[H(id_i, d') = H(id_j, d')] = 1 \neq 1/2$, and if $id_i$ and $id_j$ have different parity, then $P[H(id_i, d') = H(id_j, d')] = 0 \neq 1/2$.

Bond and McKinley also suggest using hash functions $H_{singleMult}(id, d) = \text{bit}_{31}(id \ast d)$ or $H_{doubleMult}(id, d') = \text{bit}_{31}(id \ast d^2)$. Neither function strictly satisfies the requirements, since if $id$ is small, $id \ast d$ or $id \ast d^2$ might be less than $2^{30}$ for many values of $d$, skewing the distribution of hashes to favor 0. However, especially with $H_{doubleMult}$, the skew may not be enough to matter. To fix this problem, we suggest multiplying by a random 31-bit prime number, e.g., $H_{mult}(id, d) = \text{bit}_{31}(1\,846\,571\,429 \ast id \ast d)$.

It’s clear that this hash function exhibits some periodicity, as visible in Figure 5.1, so it is not a pseudo-random function, of which MD5 [78] is an imperfect, but
better, example. Since \( \text{bit}_{31}(x) \) for a 32-bit integer \( x \) is 1 if and only if \( x/2^{32} \geq 1/2 \), we can rewrite \( H_{\text{mult}'} \) using real numbers as \( H_{\text{mult}'}(id, d) = 1 \) if and only if \( \left( \frac{1846571429\times id}{2^{32}} \right) \mod 1 \geq 1/2 \). If we instead compute \( r = \frac{2^{32}}{\pm 1846571429\times id \mod 2^{32}} \) as a real number, with the \( \pm \) chosen to ensure \( r \geq 2 \), this tells us the period of the function as \( d \) varies. For example, with \( id = 44974351 \), the randomly chosen value used for Figure 5.1, \( r \approx 6.4 \). The intuition is that if we think of \( H_{\text{mult}'}(id, d) \) as a point on a circle, each time we increase \( d \) by 1, the point rotates \( 1/r \) of the way around the circle. \( H_{\text{mult}'}(id, d) \) outputs 0 if the point is on the top half of the circle and 1 if on the bottom half. This means that \( H_{\text{mult}'}(id, d) \) for this value normally has a period of 6 (consisting of three 1 bits and three 0 bits) and sometimes has a period of 7 (consisting of at least three 1 bits and at least 3 0 bits). On one hand, the simplicity of this pattern is alarming, but on the other hand, it provides bounds on the distribution of 0 and 1 bits that \( H_{\text{mult}'} \) produces. Using just this information, we have that \( 3/7 \leq P[H_{\text{mult}'}(id, d) = 1] \leq 4/7 \) for this \( id \).

However, we can also look at the second-order periods to tighten this bound. If we look at only every sixth hop, we find that the point has rotated \( 6/r \approx 0.94 \) revolutions from where it started. Rotating \( 6/r \) revolutions in one direction is the same as rotating \( 1 - 6/r \approx 0.06 \) revolutions in the other, so after 6 hops, the point ends up a little before where it started. After \( (1/r)/(1 - 6/r) \approx 2.5 \) periods, the point has scooted back enough to fit a seventh hop in around the circle. This means that, most of the time, an extra bit follows after 5 half-periods (i.e., runs of bits of the same value) of length. This is visible in Figure 5.1a as 4 runs of length 3 being followed by a run of length 4. Since \( 2(1/r)/(1 - 6/r) \approx 4.93 < 5 \), occasionally the extra bit follows after 4 half-periods, which happens on the 9th row of Figure 5.1a, for example. This gives us a bound \( 6/13 \leq P[H_{\text{mult}'}(id, d) = 1] \leq 7/13 \). Continuing
one more step gives $122/245 \leq P[H_{\text{mult}'}(id, d) = 1] \leq 123/245$. Since $r$ is actually a rational number, only a finite number of steps (about 8) are required to completely describe the periodicity of $H_{\text{mult}'}(id, d)$. Note that although these calculations were done fixing $id$ and varying $d$, due to the symmetry of $H_{\text{mult}'}$, this is the same as fixing $d$ and varying $id$.

Taking a more theoretical approach gives a general proof of uniformity that is cleaner than iterating the above procedure many times.

**Theorem 5.1.** For any fixed $id \not\equiv 0 \pmod{2^{32}}$, $H_{\text{mult}'}(id, d) = 1$ for exactly half of the values of $d$ between 0 and $2^{32} - 1$, inclusive.

**Proof.** Fix $id$, let $C$ be the random odd prime we chose, $C = 1846571429$, and let $d$ vary from 0 to $2^{32} - 1$. We can express $id \cdot C$ as $2^s \cdot o$ for some non-negative integer $s < 32$ and an odd integer $o$. Then $id \cdot C \cdot d \pmod{2^{32}}$ is divisible by $2^s$. Thus, there’s a one-to-one correspondence between the values $(id \cdot C \cdot d) \pmod{2^{32-s}}$, which equals $o \cdot d \pmod{2^{32-s}}$, and the values $id \cdot C \cdot d \pmod{2^{32}}$ given by multiplying by $2^s$. Note that this correspondence is just a bit shift in binary. Since $o$ is odd, it is relatively prime to $2^{32-s}$ and thus invertible $\pmod{2^{32-s}}$. This means that multiplication by $o$ maps the restricted set of $0 \leq d < 2^{32-s}$ bijectively onto itself, and since exactly half of the numbers in that set have bit $(31 - s)$ equal to 1, exactly half of the outputs of $H_{\text{mult}'}$ are equal to 1 for $0 \leq d < 2^{32-s}$. Since $id \cdot C \cdot d$ is divisible by $2^s$, $id \cdot C \cdot (d + 2^{32-s}) \equiv id \cdot C \cdot d \pmod{2^{32}}$, so $H_{\text{mult}'}(id, d) = H_{\text{mult}'}(id, d + 2^{32-s})$, which means that the uniformity extends when $0 \leq d < 2^{32} - 1$. 

We were hoping that the following theorem would be true:

**Theorem 5.2.** For any fixed $id_1, id_2$ with $id_1 \not\equiv \pm id_2 \pmod{2^{32}}$, $H_{\text{mult}'}(id_1, d) = H_{\text{mult}'}(id_2, d)$ for exactly half of the values of $d$ between 0 and $2^{32} - 1$, inclusive.
However, we found that it was false. Furthermore, the characterization of independence used in Theorem 5.2 is actually weaker than the full independence required for the sum of Bernoulli trials to follow a binomial distribution.

As a concrete counterexample to Theorem 5.2, set \( id_1 = 2 \) and \( id_2 = 6 \). Enumerating all \( 2^{32} \) values of \( d \) reveals that \( H_{mult'}(2, d) = H_{mult'}(6, d) \) is true for \( 2/3 \) of the values of \( d \) instead of \( 1/2 \).

Because of symmetry, there is also correlation between \( H_{mult'}(id, 2) \) and \( H_{mult'}(id, 6) \), which means that congestion on link 2 will cause link 6 to appear congested, and vice versa.

In practice, this problem seems limited to when one of the arguments is small. However, we can ensure \( d \) is never small by carefully choosing how the integer \( d \) is computed from the switch ID and port number. We propose \( d = (\text{switch ID}) \ast 2^{15} + (\text{port number}) \ast 2^9 + 509 \), since 509 is the largest prime that fits in 9 bits. This ensures that \( d \) is always greater than 500, which seems to be large enough.

The cost of computing \( H_{mult'} \) is much cheaper than computing a cryptographic hash, so we propose using \( H_{mult'} \) in spite of it not being a pseudo-random function. Another possibility is to compute the least significant bit of the CRC of \( id \ast d \). Switches already contain hardware for fast CRC calculations over packets, and this function appears to perform well experimentally.

### 5.5 Experimental analysis of probabilistic approach

We evaluate the probabilistic scheme on the naïve reduction traffic pattern, since it exhibits a variety of congested fraction values.

Figure 5.2 shows the same network traffic as Figure 4.12, but computed using \( H_{mult'} \) and \( d \) defined as above. Notice that it fairly faithfully reproduces the congestion.
Figure 5.2: Congested fraction plot generated using the probabilistic variant of our reservoir sampling-based scheme of the naïve reduction to rank 0 with 6.5 million packets.

Figure 5.3: Congested fraction plot generated using the probabilistic variant of our reservoir sampling-based scheme of the naïve reduction to rank 0 simulating 10 times as many packets but the same total communication volume.
of the links with the largest amount of traffic, and there are no false positives, but many congested links through with fewer packets traveled do not appear. This uses a false positive threshold $T$ set to hide any links for which the probability of observing $W_d \geq T$ is less than 99.5% given that the actual number of packets that passed through $d$ was 0. Increasing the number of packets, as in Figure 5.3, which has 10 times as many packets, provides accurate data about more links without increasing the number of false positives. To ensure a fair comparison between Figure 5.2 and Figure 5.3, the same volume of traffic is generated in each experiment. Even with the original number of packets, the problem of endpoint congestion at rank 0 is apparent.

As discussed in section 5.2, only updating the count for the subset of the links through which a packet might possibly have passed based on the source and destination of the packet dramatically improves our ability to discriminate congestion from noise due to the randomness in the algorithm. We repeated the same experiments as Figure 5.2 and Figure 5.3 using a version of the algorithm that implements this opti-
Figure 5.5: Congested fraction plot generated using the optimized probabilistic variant of our reservoir sampling-based scheme of the naïve reduction to rank 0 simulating 10 times as many packets but the same total communication volume.

5.6 Evaluating the storage requirements of the probabilistic approach

Recall that the end goal of the probabilistic method is to enable reservoir sampling to fit in the packet header. Using InfiniBand as an example, we should examine the header more closely to see exactly how many bits we can use.

All InfiniBand packets contain at least the local routing header (LRH) and base transport header (BTH). Currently, the LRH and BTH contain two reserved fields each, of width 2 and 5 bits for the LRH, and 6 and 7 bits for the BTH. Even if all 20 bits were allocated to storing reservoir sampling information, this would not be
enough to store all the fields we need without the probabilistic method. Exacerbating this problem is the fact that all reserved fields except the 6 bit field are included in the invariant CRC computation. This means that those fields cannot be modified after the packet is generated, i.e., cannot be modified by a switch, without breaking compatibility with existing InfiniBand switches.

Realistically, it is possible to reduce the number of bits used to store the hop count from 8. If the count is incremented using saturating arithmetic, paths longer than \(2^b - 1\) introduce additional error, where \(b\) is the number of bits used to store the hop count; however, these paths do not cause catastrophic errors in the algorithm like a count rolling over would. For example, suppose \(b = 2\), and a packet is travelling along a path of length 5. The algorithm proceeds normally on the first three switches, and the only difference at the fourth switch is that the hop count remains at 3. The fifth switch reads the hop count of 3, so it replaces the reservoir with probability \(1/4\) instead of \(1/5\). When the packet arrives at a compute node, the reservoir will contain information from the fifth hop with probability \(1/4\) and from the first four hops each with probability \(3/16\). Furthermore, the compute node will also under-weight the packet because the count is lower than the number of traversed hops, resulting in expected weighted counts of \(9/16, 9/16, 9/16, 12/16\), when the correct values are 1, 1, 1, 1, 1. Every hop is under-counted, and the distribution is no longer uniform.

Paths of length 5 are common, and paths of length 7 seem reasonable for hierarchical networks (e.g., fat tree and dragonfly), so we need \(b \geq 3\). Paths on grid- or torus-based networks are much longer, so \(b = 4\) might be a better choice for those networks.

Even using the probabilistic scheme needs at least 4 variant bits—a 1 bit hash and 3 bit path length—for the normal reservoir sampling, and another 4 variant bits for
the congested reservoir sampling. Although this is a substantial reduction from the storage requirements of the non-probabilistic scheme, it is still too large to achieve our design goal.

5.7 An even smaller variant

Instead of using a parallel reservoir for the congested hops, the packet can contain one bit indicating whether the output port stored in the normal reservoir was congested. This brings the total requirement down to 5 or 6 bits, which fits in the current InfiniBand header. The pseudocode for this variant appears in algorithm 8 and algorithm 9.

Algorithm 8: Reservoir Sampling for Understanding Congestion with One Reservoir: On every switch

**Input:** The current packet, \( \text{packet} \)

1. \( \text{outport} \leftarrow \) the port through which \( \text{packet} \) will leave the current switch
2. \( \text{rand1} \leftarrow \) a random integer between 0 and \( \text{packet.hop\_count} \), inclusive
3. if \( \text{rand1} = 0 \) then
   4. \( \text{packet.hop\_bit} \leftarrow H(\text{packet.id}, (\text{my switch ID}, \text{outport})) \)
   5. if \( \text{outport is congested} \) then
      6. \( \text{packet.congested\_bit} \leftarrow 1 \)
   7. else
      8. \( \text{packet.congested\_bit} \leftarrow 0 \)
8. \( \text{packet.hop\_count} \leftarrow \text{packet.hop\_count} + 1 \)
Algorithm 9: Reservoir Sampling for Understanding Congestion with One Reservoir: On the receiver NIC

Input: The received packet, \( \text{packet} \); maps \( \text{counts} \) and \( \text{congested\_counts} \) from the set \( D \) of tuples (switch ID, output ports) to integers representing an estimate of the number of packets (respectively packets that encountered congestion) that passed through the specified output port. Both maps should be initialized to 0 prior to receiving any packets.

1. if \( \text{packet.hop\_count} > 0 \) then
2.  foreach \( d \in D \) where \( \text{packet might have passed through } d \) do
3.    if \( H(\text{packet.id}, d) = \text{packet.hop\_bit} \) then
4.      \( \text{counts}[d] \leftarrow \text{counts}[d] + \text{packet.hop\_count} \)
5.      if \( \text{packet.congested\_bit} = 1 \) then
6.        \( \text{congested\_counts}[d] \leftarrow \text{congested\_counts}[d] + \text{packet.hop\_count} \)
7.    else
8.      \( \text{counts}[d] \leftarrow \text{counts}[d] - \text{packet.hop\_count} \)
9.      if \( \text{packet.congested\_bit} = 1 \) then
10.     \( \text{congested\_counts}[d] \leftarrow \text{congested\_counts}[d] - \text{packet.hop\_count} \)

5.8 The Bloom filter perspective

Another common probabilistic strategy for storing a large set of values with a few bits is a Bloom filter [79]. A Bloom filter is a \( m \)-bit array with \( k \) hash functions from \( D \rightarrow \{0, 1, \cdots, b - 1\} \). To insert an element \( d \in D \) into the Bloom filter, the \( h(D)^{th} \)
bit in the array is set to 1 for each of the \( k \) hash functions \( h \). One significant difference between a Bloom filter and explicitly storing a \( m \)-bit hash of \( d \) is that Bloom filters can store multiple elements simultaneously. Inserting switch IDs and port IDs into a Bloom filter stored in the packet, gives us an alternative method to reservoir sampling for storing a variable amount of information in a fixed amount of space.

However, storing a single \( m \)-bit hash explicitly is equivalent to using a \( 2^m \)-bit Bloom filter with a single hash function. In particular, when \( m = 1 \), the correspondence is \( \text{packet.hop.bit} = 0 \Leftrightarrow [1,0] \), and \( \text{packet.hop.bit} = 1 \Leftrightarrow [0,1] \). The two possible states of a two-bit Bloom filter with one hash function after inserting one element are exactly the two states in the correspondence.

Even with more than one hash function, a Bloom filter for a single element has a higher false positive rate than explicitly storing a hash of the same size. The probability of a false positive from an \( m \)-bit Bloom filter with one item, \( FP_{\text{Bloom}} \), is approximately \((1 - e^{-k/m})^k\) [80]. This is minimized when \( k = m \log 2 \) [80], so

\[
FP_{\text{Bloom}} \geq (1 - e^{-\log 2})^{m \log 2} = \left(\frac{1}{2}\right)^{m \log 2} \geq \left(\frac{1}{2}\right)^m = FP_{\text{Hash}},
\]

where \( FP_{\text{Hash}} \) is the probability of a false positive when storing a \( m \)-bit hash explicitly. Thus, in the case in which only one element will be stored in the Bloom filter, storing the hash explicitly achieves a lower false positive probability with the same number of bits.

### 5.9 Generating random numbers

Our reservoir sampling scheme in any form requires generating random integers between 0 and \( l \) on the switch, for various values of \( l \), a small integer that may not be a power of 2. For the mathematical error analysis to be sound, these numbers
must follow a uniform distribution, and each random number should be statistically independent from the others.

One technique for rapidly generating high quality random numbers in hardware uses self-oscillating circuits with metastable latches as a source of entropy, as implemented for the RDRAND instruction in Intel CPUs [81]. The latch oscillates based on thermal noise at approximately 3 GHz, and is sampled at 800 MHz. Although the bit stream from the circuit contains true randomness, it still needs conditioning before it is usable as unbiased random numbers. The entropy is conditioned with the AES encryption algorithm.

Since reservoir sampling does not require cryptographically secure random numbers, we can use a less complex scheme. Additionally, since we only use reservoirs of size 1, we only check whether the randomly generated number is equal to 0 or not, so instead of generating a full integer, it suffices to generate a single bit that is 0 with probability $1/l$. We suspect that a simple $l$-bit shift register (see Figure 5.6) with one all but one bit initialized to 1 clocked at a different frequency would suffice. Clearly this circuit does not produce random numbers, but clocked at several gigahertz, perhaps with a self-oscillating latch, and only sampled once per packet, we suspect that the output of the shift register would be indistinguishable from a true random source. The switch would need one shift register for each possible path length, but given how simple each register is, we do not believe this is a challenge. To ensure that a stable value is read from the last multiplexer, the clock should be suppressed by asserting the signal labeled Read prior to reading a value from Output.

Another approach is to use a normal linear feedback shift register [82] to generate $b$ bits and to compare it with an approximation of $(2^b - 1)/l$ stored in a pre-programmed lookup table, as shown in Figure 5.7. When $(2^b - 1)/l$ is not an integer, the probability...
Figure 5.6: A 3-bit shift register that will output 0 1/3 of the time.

Figure 5.7: A circuit that generates an output of 0 approximately 1/l of the time.

will not be exact, but will be within 1/(2(2^b - 1)) of 1/l. When doing the analysis, it’s important to remember that linear feedback shift registers never output the value 0. If chosen correctly the LFSR has a period of 2^b - 1, so each non-zero possible value is reached with uniform probability.

Finally, another alternative approach is to generate random bits with the required distribution at the NIC prior to sending the packet. These random values can be stored in a field in the packet itself. Instead of generating a random number, the
switch simply reads the leading bit in the field and then shifts the field 1 bit to the left, consuming and discarding the leading bit. At a cost of one additional bit, the sending NIC can insert a 1 bit at the end of the field, which then allows the receiving NIC to see how many times the field was left-shifted. This eliminates the need for a separate hop count field, which then does not need to be incremented at a switch. Storing the bits explicitly replaces a field of size $\log_2 L$ bits with a field of size $L + 1$ bits, where $L$ is the maximum possible path length.

### 5.10 Identifying congestion

Several design decisions must be made to determine precisely what it means for an output port to be congested. The definition we used for congestion in our experiments is this: an output port is congested when the length of its queue exceeds its available credits. This means a congested port must receive credit back before it can empty its output queue, though it does not necessarily mean that the port will run out of credits and stall. An output port could also be considered congested when the length of its queue exceeds some threshold, or when its available credit falls below a threshold.

An interesting alternative approach is to use a probabilistic scheme, where packets are marked as congested with some probability based on how congested the output port is. This eliminates the need for a strict threshold and also allows reservoir sampling to capture richer information about the level of congestion.

Additionally, there are at least three potential times when congestion can be measured for a switch with both input and output queues: when a packet arrives, as a packet is inserted into an output queue, and immediately prior to the packet leaving the switch. In an uncongested switch, the time difference between these three points is minuscule, but in a heavily congested switch, they could differ by several
microseconds. This choice will change exactly which packets are marked as congested, but is unlikely to dramatically change the overall counts and subsequent analysis. Unless congestion lasts less than the time to transmit a few packets (hundreds of nanoseconds), any choice of time will detect it.

- At the exact instant an output port becomes congested, there will be a packet in the output queue, since if the output queue were empty, it would not be congested.

- Very near to that instant, new packets for the congested output port will arrive, since if they don’t arrive, the congestion will subside quickly.

- As long as new packets are arriving, packets are also being inserted into the output queue.

So, all three times for measurement will have the opportunity to mark packets as congested.

The pragmatic resolution to these design decisions is to use the existing mechanisms for congestion control. Since the congestion control parameters are provisioned by the system administrators and switch manufacturer, this also makes the definition of congestion more flexible. The main benefit, however, is that it reduces the complexity of implementing reservoir sampling, since new circuitry to detect congestion need not be designed.
Chapter 6

Conclusion and future work

We presented a new technique for understanding the communication performance of applications. This technique uses reservoir sampling to store a small amount of information in the packet about the path taken by the packet and any congestion encountered. From this information, collected in many packets, we can determine whether a communication performance problem stems from a poor communication pattern, a poor mapping, or contention for shared networking resources with background traffic from other applications. Each root cause has a different remediation strategy, so distinguishing between them is useful.

Our reservoir sampling scheme is packet-centric, which means that the data it collects can be combined with the application contexts in which a message was sent and received, to provide a code-level view of communication performance. At the same time, our scheme is also link-centric, which means that the information we collect about the network is at the granularity of a single link. Besides collecting raw bandwidth statistics, we also collect information about whether a link is congested, allowing us to distinguish between full utilization of a link, which is desirable, and actual congestion, which is not.

Code-level profiling or tracing tools, like those that use the P-MPI profiling interface to measure communication costs can attribute measurements to a line of code but provide little insight as to why communication might be slow. Tools that only provide a network-level view can attribute communication performance problems to
the actual network, but increase the difficulty of fixing a problem that occurs in a large application. Our technique can combine the insights of both types of tools. Furthermore, our scheme automatically enables analysis at an extremely fine-grained time analysis, which is useful for observing how congestion patterns emerge and evolve over time.

A slightly modified form of our reservoir sampling scheme is potentially practical to implement in networking hardware with negligible performance impact. Our scheme does not require additional buffers on the switches for recording performance events, and only a small amount of additional logic per output port. The variant detailed in section 5.7 is even compatible with the InfiniBand specification and uses only 5 or 6 bits per packet. Because those currently unused bits are transmitted in the packet, this version does not create any additional bandwidth overhead.

In chapter 4, we evaluate measurements gathered using our reservoir sampling scheme for several synthetic traffic patterns. Applying our reservoir sampling scheme to measure a naïve reduction communication pattern in which all nodes send their inputs to a single node, enables diagnosis of the underlying performance problem: a poor communication pattern. Rewriting the code to use a more scalable reduction strategy obtained a nearly 90x performance improvement.

Second, we apply our technique to understand congestion for a two dimensional stencil communication pattern. Reservoir sampling reveals that the root cause is a poor mapping between the logical and physical topology, and in particular, that the primary bottleneck is bandwidth between the leaf and aggregation switches. Targeted fixes to the mapping designed to address this bottleneck with tiling or using graph partitioning algorithms exhibit over 25% performance improvement over the default row major mapping, while generic topology-oblivious strategies do not address the
topology-specific bottleneck and worsen performance.

Third, we look at a communication pattern that exhibits run-to-run variations of 54%. Even without privileged network-administrator level information, we are able to diagnose conclusively that the performance variation is due to background traffic running at the same time. As proposed in the thesis statement, in each of these three situations, our reservoir sampling scheme provides a powerful and practical approach for understanding network traffic and congestion. The rules in section 4.1 guide the application developer from measurements collected with reservoir sampling to a diagnosis of the underlying communication performance problem.

6.1 Future directions

The next step is to experiment with using reservoir sampling for measurement and diagnosis of communication performance problems in more realistic applications. The synthetic patterns used in this thesis are relatively simple compared to real applications. The simplest way to achieve this is by collecting networking traces and replaying them in the CODES network simulator. Even though CODES may not be able to simulate application performance with perfect fidelity, if our reservoir sampling scheme is able to diagnose communication performance problems in simulation and lead to fixes, it strongly suggests that it will also be able to diagnose communication performance problems in real hardware.

Although communication mapping has been studied before, dynamic information about the real-time status of the network collected using reservoir sampling could enable a new class of algorithms. In particular, we believe that incremental hierarchical graph clustering algorithms, when paired with edge weights derived from real-time information about the cost of network paths, could result in topology-aware
and background traffic-aware mappings that outperform static mappings.

We would also like to extend the analysis of the modification suggested in section 5.7. These modifications, unlike others, only make use of one reservoir, which means that the statistical analysis is somewhat different. Accurately characterizing the statistical properties of the scheme is important before implementing it.

Since the end goal is to diagnose communication performance problems of real applications on real systems and not in a simulator, our reservoir sampling scheme ultimately must be implemented on real hardware to make a meaningful impact. We believe not only that this is practical, but also, and more importantly, that the information collected by this scheme, even if just a few bits per packet, can be immensely valuable to HPC application developers in helping them improve communication performance.
Bibliography


[40] “UFM demo with Jeff Margolis @ SC12.” https://www.youtube.com/watch?v=8Cr0ZeXsnGQ, 2012.


