# NETWORK FLOW OPTIMIZATION FOR DISTRIBUTED CLOUDS

BY

SANGEETHA ABDU JYOTHI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Associate Professor Brighten Godfrey, Chair
Associate Professor Matthew Caesar
Professor Klara Nahrstedt
Professor Jennifer Rexford, Princeton University
Professor Rayadurgam Srikant

# ABSTRACT

Internet applications, which rely on large-scale networked environments such as data centers for their back-end support, are often geo-distributed and typically have stringent performance constraints. The interconnecting networks, within and across data centers, are critical in determining these applications' performance. Data centers can be viewed as composed of three layers: physical infrastructure consisting of servers, switches, and links, control platforms that manage the underlying resources, and applications that run on the infrastructure. This dissertation shows that network flow optimization can improve performance of distributed applications in the cloud by designing high-throughput schemes spanning all three layers.

At the physical infrastructure layer, we devise a framework for measuring and understanding throughput of network topologies. We develop a heuristic for estimating the worst-case performance of any topology and propose a systematic methodology for comparing performance of networks built with different equipment. At the control layer, we put forward a source-routed data center fabric which can achieve near-optimal throughput performance by leveraging a large number of available paths while using limited memory in switches. At the application layer, we show that current Application Network Interfaces (ANIs), abstractions that translate an application's performance goals to actionable network objectives, fail to capture the requirements of many emerging applications. We put forward a novel ANI that can capture application intent more effectively and quantify performance gains achievable with it.

We also tackle resource optimization in the inter-data center context of cellular providers. In this emerging environment, a large amount of resources are geographically fragmented across thousands of micro data centers, each with a limited share of resources, necessitating cross-application optimization to satisfy diverse performance requirements and improve network and server utilization. Our solution, Patronus, employs hierarchical optimization for handling multiple performance requirements and temporally partitioned scheduling for scalability.

*To my family and friends*

# ACKNOWLEDGMENTS

I am extremely grateful to my advisor, Brighten Godfrey, for his guidance and support throughout my Ph.D. I learned several invaluable skills from him including finding the right problems, distilling a problem to its essence, and presenting ideas in a clear and concise manner. Many a time I have walked into his office feeling dejected about research progress, but I *always* left with an optimistic spirit, buoyed by his positivity and his ability in getting to the core of a problem by asking the right questions. I believe that his steady support and generosity also played a key role in preserving my love for research while going through the vagaries of graduate school. Brighten has been a truly inspiring role model as a researcher, teacher, mentor, and as a person. Thanks for this privilege, Brighten.

I would like to thank my dissertation committee members, Matthew Caesar, Klara Nahrstedt, Jennifer Rexford, and Rayadurgam Srikant, for providing valuable feedback that helped in shaping this dissertation. During my Ph.D., I also had the privilege to work with several amazing researchers. Aditya Akella helped in honing the Patronus project with his invaluable insights and feedback. I am also thankful to him for his guidance during the job search. I would like to express my sincere gratitude to Ishai Menache who took me as an intern during the early phase of my Ph.D., guided me through completion of an exciting project, and continues to look out for my career ever since. I am immensely grateful for his unwavering support, guidance, and friendship. Ankit Singla has been a great mentor, collaborator, and friend beginning from my early days at the University of Illinois, Urbana-Champaign (UIUC). His guidance proved to be crucial for the completion of my first project at UIUC.

I am thankful to my amazing collaborators who contributed to various parts of this thesis. Ankit Singla and Alexandra Kolla helped in shaping the work on measuring and understanding throughput (Chapter 2). Mo Dong provided valuable help for developing source routed data center fabric (Chapter 3). In addition to guidance from Aditya Akella, Patronus (Chapter 4) has benefited from experiments done by an enthusiastic undergrad I mentored, Ruiyang Chen. Finally, I am indebted to Sayed Hadi Hashemi for introducing me to the area of machine learning systems. A random conversation in a hallway led to a series of work on accelerating distributed deep learning with network scheduling. CadentFlow (Chapter 5), in collaboration with Hadi and Roy Campbell, is a part of this line of work. I am also thankful to Carlo Curino, Subru Krishnan, and their team for their mentoring and support during my Microsoft internship. The hands-on experience I gained with this team working on a large-scale system has helped me tremendously in my other projects.

I would like to thank Indranil Gupta for helping with every phase of the job search and for serving on my preliminary examination committee. I am grateful to Tianyin Xu for sharing in detail his recent experiences on the market and for patiently correcting multiple versions of my job talk. Special thanks to Darko Marinov for running the Pilot series of talks which proved very helpful. I would also like to express my gratitude to Matus Telgarsky for being a great friend, sharing a realistic view of the academic world, and helping with the interview process.

I would like to extend my gratitude to Roch Guerin and Boon Thau Loo at the University of Pennsylvania who guided me during my Masters thesis and faculty members at the National Institute of Technology, Calicut, my undergrad alma mater, for their continuing support and encouragement.

I am thankful to the National Science Foundation and Facebook for funding my work.

Special thanks to all administrative staff who served in the Computer Science Department over the past six years, for helping with every difficulty in a timely manner.

Conversations with my colleagues in the networking and systems lab have greatly helped my work, thanks for an intellectually stimulating and fun work environment, and a pleasant grad school experience — Rachit Agarwal, Jason Croft, Mo Dong, Fred Douglas, Soudeh Ghorbani, Mainak Ghosh, Vipul Harsh, Chi-Yao Hong, Virajith Jalaparti, Nathan Jay, Faria Kalim, Ahmed Khurshid, Qingxi Li, Bingzhe Liu, Chia-Chi Lin, Tong Meng, Santhosh Prabhu, Ankit Singla, Rashid Tahir, Ashish Vulimiri, Anduo Wang, Wenxuan Zhou.

During the past six years, Champaign has been a home away from home because of my amazing friends. Thank you, Anand, Anjali, Aswin, Bhargava, Faria, Hadi, Jacob, Kaushik, Kavya, Kiran, Kishor, Krishna, Mahesh, Mainak, Maneesha, Manu, Pavithra, Prasanna, Rachana, Sahand, Sandeep, Sandhya, Santhosh, Sarath, Shalmoli, Sinduja, Sooraj, Sridhar, Subhro, Sushma, Virajith, and Vivek — for potlucks, escape rooms, roadtrips, game nights, and all the wonderful times we spent together. A special thanks for your kindness and support during my lows. Although miles away, I am also grateful for nearly decade-long unwavering friendship of Nidhin, Sananda, and Vaidyanathan.

Most importantly, I would like to thank my family for their unconditional love and support. My parents, Abdu T. K. and Jyothi K. C., have always sacrificed their own needs to provide me with a platform to succeed. I owe everything to their love, commitment to my happiness, and encouragement towards my pursuits while bearing with my stubbornness. I am also immensely inspired by their dedication to both family and work. Finally, my brother, Akash, has had a profound influence on my intellectual and personal growth with his calm and thoughtful demeanor, and his deep insights during our long conversations. Thanks for being the yin to my yang. I dedicate this dissertation to my family and friends.

## TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1: INTRODUCTION

Data centers are critical in providing back-end support for most Internet applications today. Applications running in these large-scale networked environments range from data-intensive [2, 3, 4] to user-facing online services (web browsing, real-time streaming applications, etc.). These applications often have stringent performance requirements in terms of latency, throughput, etc. When components of such applications are distributed within and across multiple clouds/data centers, their performance is determined by the underlying infrastructure. In particular, the interconnecting network within and across data centers directly impacts the application performance.

In order to design high-throughput solutions for intra- and inter-data center networks, first, it is necessary to understand roadblocks in achieving high performance at various layers in these networks. The data center infrastructure can be viewed as composed of three layers: (a) the physical infrastructure composed of servers, switches, and links, (b) control platforms that manage the underlying physical resources, and (c) applications that run on the infrastructure. Each layer presents unique challenges.

At the physical layer, network throughput is limited by the carrying capacity of the interconnect: the network topology and link speeds. In this layer, the challenge revolves around understanding the throughput limits of a given topology including traffic patterns that trigger bad performance. At the control layer, the control schemes are responsible for narrowing the gap between the optimal achievable throughput on a given interconnect and the actual throughput. The challenge at this layer can be mapped to leveraging the large number of available paths while using minimal memory overhead at switches and processing overhead across the network. At the application layer, the key hurdle is translating higher level application performance goals to actionable network objectives which can be handled by a network controller.

This dissertation shows that network flow optimization can optimize performance of distributed applications in the cloud by designing high-throughput schemes spanning all three layers, within and across data centers. First, at the physical layer, we study the limits of achievable throughput in networks. Second, for efficient intra-data center control, we propose a source-routed data center fabric. Third, for inter-data center control, we design Patronus which provides scalable resource optimization across geo-distributed micro data centers. Fourth, at the application layer, we develop a new Application Network Interface (ANI) for effectively translating high-level application performance objectives to actionable network objectives.

At the physical layer, we gain an in-depth understanding of the fundamental limits of network throughput [5]. We analyze whether commonly-used cut-based metrics, such as bisection bandwidth and sparsest cut, solve the problem of estimating worst-case throughput. We show that they do not, both theoretically and using a simple example of a highly-structured network of small size. Since cut metrics do not achieve our goal, we develop a heuristic to measure worst-case throughput. We also build a framework to perform a head-to-head benchmark of a wide range of topologies across a variety of workloads [6], revealing insights into the performance of topologies with scaling, robustness of performance across Traffic Matrices (TMs), and the effect of scattered workload placement.

At the control layer in the intra-data center environment, we propose an efficient solution for a network fabric, where complex application-sensitive functions are factored out, leaving the network itself to provide a simple, robust high-performance data delivery abstraction. This requires performing route optimization, in real time and across a diverse choice of paths. A large variety of techniques have been proposed to provide path diversity for network fabrics. But, running up against the constraint of forwarding table size, these proposals are topology-dependent, complex, and still only provide limited path choice which can impact performance. We propose a simple approach to realize the vision of a flexible, high-performance fabric: the network should expose every possible path, allowing a controller or edge device maximum choice. To this end, we observe that source routing can be encoded and processed compactly into a single field, even in large networks. We show that, in addition to the expected decrease in required forwarding table size, source routing supports optimal throughput performance.

In the emerging area of geo-distributed micro data centers, this dissertation focuses on control of thousands of data centers and the interconnecting Wide Area Network (WAN) for improving application performance. This environment is composed of multiple constrained and expensive entities: (a) wide area links with limited bandwidth and (b) micro data centers with a few racks of servers. In order to meet the goals of geo-distributed applications while simultaneously utilizing resources efficiently, we need an efficient resource management scheme. However, this is difficult for several reasons. First, the combination of scale and geographic spread has not been addressed by prior large-scale systems. Second, the environment needs to support a motley set of applications with diverse requirements. This includes long-running streaming applications (e.g., cellular Virtualized Network Functions (VNFs), other middlebox service chains), batch analytics (e.g., cellular log analytics, Hadoop jobs) and Lambda-like short-lived jobs (e.g., elastic web servers). To meet the requirements of such geo-distributed high-performance applications, resource allocation on distributed Micro Data Centers (MDCs) and the interconnecting WAN will need to be coordinated. Third, the smaller size of MDCs and the potential for demand bursts mean that resource availability in any particular MDC will be more dynamic and variable than in a hyperscale DC. In other words, MDCs enjoy limited benefits of statistical multiplexing. Thus, this environment is characterized by its *scale*, *geographic spread*, *diversity of applications*, and *limited resources at MDCs*. While one or two of these challenges have been addressed in existing large-scale systems [7, 8, 9, 10, 11], the combination of all characteristics calls for novel resource management techniques in WAND. We design Patronus for efficient resource control of geo-distributed micro data centers interconnected by a WAN.

At the application layer, we show that state-of-the-art application network interface (ANI), the abstraction that translates an application's performance intent to network objectives which can be achieved by a network controller, fail to capture the requirements of many emerging applications. ANIs and the flexibility they offer have evolved over time. The earliest congestion control and traffic engineering schemes focused on simple proxies for application performance at *packet* level — throughput and per-packet delay and jitter. Rate Control Protocol [12] made a step towards application-level performance goals with *flow* as the ANI and emphasis on Flow Completion Time (FCT) or the time of arrival of the last packet. Another significant leap towards an ANI that captures the requirements of distributed applications was the *coflow* [13]. Inspired by cloud applications such as MapReduce, coflow considers a set of parallel flows within an application as a single entity where the FCT of the last flow determines the performance. This enables scheduling schemes to borrow bandwidth from lighter flows in the coflow to speed up the heavier flows, thereby improving application deadlines. We observe that even the coflow abstraction is insufficient to support requirements of today's sophisticated applications. Applications such as distributed deep learning and interactive analytics have a complex interplay of communication and computation at the participating nodes. In this scenario, not all flows within a coflow are equivalent from the perspective of the application. Depending on the nature of computation, the application may benefit by finishing some flows sooner than others within a coflow. We quantify the performance loss accrued by current application network interfaces and put forward a novel interface that can capture application intent more effectively.

### Thesis Roadmap

Chapter 2 presents the work on physical infrastructure level analysis of measuring and understanding throughput of network topologies. Results in this chapter appeared in [5]. Chapter 3 includes the proposal on a source-routed data center fabric that can achieve near-optimal throughput performance using minimal memory at network switches. These results were presented in [14]. Chapter 4 presents the control challenges in the emerging environment of geo-distributed micro data centers and our solution for resource management in this environment, Patronus. Chapter 5 explores the shortcomings of current Application Network Interfaces (ANIs), quantifies the performance loss due to the inability of state-of-the-art ANIs, and puts forward a new ANI, CadentFlow.

# Chapter 2: MEASURING AND UNDERSTANDING THROUGHPUT

In data center and high performance computing environments, an increase in throughput demand among compute elements has reinvigorated research on network topology. Although a large number of network topologies have been proposed in the past few years to achieve high capacity [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26], a systematic methodology for measuring and comparing throughput performance of topologies has been conspicuously absent. In this work, we analyze current approaches used in measuring throughput and build a framework for throughput measurement and comparison across topologies.

## 2.1 MEASURING THROUGHPUT

The metric of interest is end-to-end throughput supported by a network in a fluid-flow model with optimal routing (not considering higher-level design like routing protocols and congestion control).

### 2.1.1 Defining Throughput

A **network** is a graph $G = (V, E)$ with capacities $c(u, v)$ for every edge $(u, v) \in E_G$. Among the nodes $V$ are **servers**, which send and receive traffic flows, connected through non-terminal nodes called **switches**. Each server is connected to one switch, and each switch is connected to zero or more servers, and other switches. Unless otherwise specified, for switch-to-switch edges $(u, v)$, we set $c(u, v) = 1$, while server-to-switch links have infinite capacity. This allows us to stress-test the network topology itself, rather than the servers.

A **traffic matrix (TM)** $T$ defines the traffic demand: for any two servers $v$ and $w$, $T(v, w)$ is an amount of requested flow from $v$ to $w$. We assume without loss of generality that the traffic matrix is normalized so that it conforms to the "hose model": each server sends at most 1 unit of traffic and receives at most 1 unit of traffic ($\forall v$, $\sum_w T(v, w) \leq 1$ and $\sum_w T(w, v) \leq 1$).

The **throughput** of a network $G$ with TM $T$ is the maximum value $t$ for which $T \cdot t$ is feasible in $G$. That is, we seek the maximum $t$ for which there exists a feasible multicommodity flow that routes flow $T(v, w) \cdot t$ through the network from each $v$ to each $w$, subject to the link capacity and the flow conservation constraints. This can be formulated in a standard way as a linear program (omitted for brevity) and is thus computable in polynomial time. If the nonzero traffic demands $T(v, w)$ are equal, this is equivalent to the *maximum concurrent*

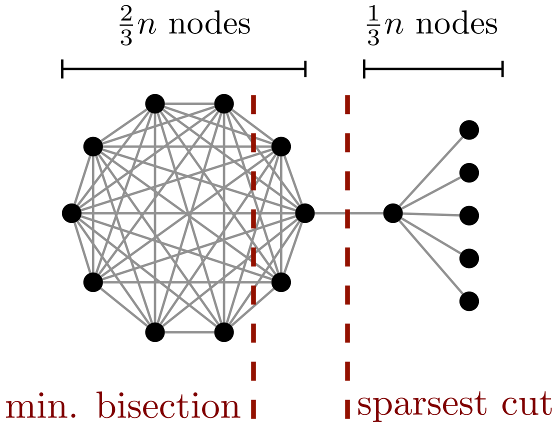


Figure 2.1: Bisection bandwidth fails to find the true bottleneck.

*flow problem* [27]: maximizing the minimum throughput of any requested end-to-end flow.

### 2.1.2 Cuts: a weak substitute for worst-case throughput

Cuts are generally used as proxies to estimate throughput. Since any cut in the graph upper-bounds the flow across the cut, if we find the minimum cut, we can bound the worst-case performance. Two commonly used cut metrics are:

*(a) Bisection bandwidth:* It is a widely used to provide an evaluation of a topology's performance independent of a specific TM. It is the capacity of the worst-case cut that divides the network into two equal halves ([28], p. 974).

*(b) Sparsest cut:* The sparsity of a cut is the ratio of its capacity to the net weight of flows that traverse the cut, where the flows depend on a particular TM. Sparsest cut refers to the minimum sparsity in the network. The special case of *uniform sparsest cut* assumes the all-to-all TM.

Cuts provide an upper-bound on worst-case network performance, are simple to state, and can sometimes be calculated with a formula. However, they have several limitations.

**(1) Bisection bandwidth does not always capture the worst-case cut:** The insistence on splitting the network in half means that bisection bandwidth may not uncover the true bottleneck. In the graph $G$ of Figure 2.1, any bisection must split off at least $\frac{1}{6}n$ of the nodes in the large clique, and each of these have $\Theta(n)$ neighbors, meaning that $BB(G) = \Theta(n^2)$ while the graph actually has a bottleneck consisting of just a single link!

**(2) Sparsest cut and bisection bandwidth are not actually TM-independent**, contrary to the original goal of evaluating a topology independent of traffic. Bisection bandwidth

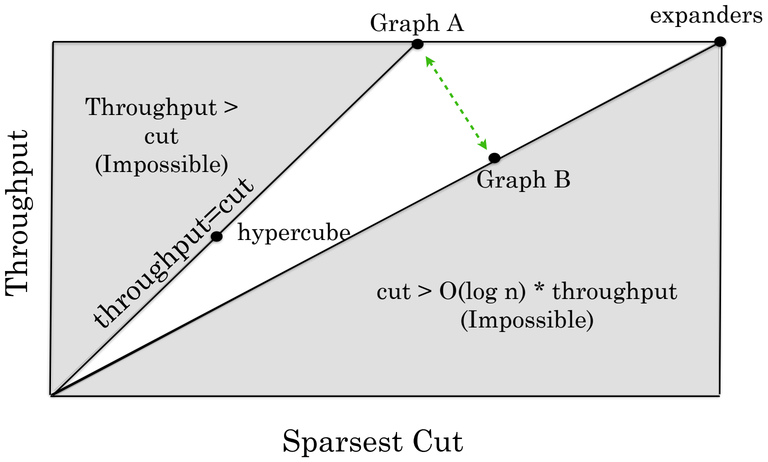

Graph A
expanders
Throughput > cut (Impossible)
throughput=cut
hypercube
Graph B
cut > O(log n) * throughput (Impossible)
Throughput
Sparsest Cut

Figure 2.2: Sparsest cut vs. Throughput

and the uniform sparsest cut correspond to the worst cuts for the complete (all-to-all) TM, so they have a hidden implicit assumption of this particular TM.

**(3) Even for a specific TM, computing cuts is NP-hard**, and it is believed that there is no efficient constant factor approximation algorithm [29, 30]. In contrast, throughput is computable in polynomial time for any specified TM.

**(4) Cuts are only a *loose* upper-bound for worst-case** throughput. This may be counter-intuitive if our intuition is guided by the well-known max-flow min-cut theorem which states that in a network with a single flow, the maximum achievable flow is equal to the minimum capacity over all cuts separating the source and the destination [31, 32]. However, this no longer holds when there are more than two flows in the network, i.e., *multi-commodity flow*: the maximum flow (throughput) can be an $O(\log n)$ factor lower than the sparsest cut [33]. Hence, cuts do not directly capture the maximum flow.

Figure 2.2 depicts this relationship between cuts and throughput. Here we focus on sparsest cut.[1] The flow (throughput) in the network cannot exceed the upper bound imposed by the worst-case cut. On the other hand, the cut cannot be more than a factor $O(\log n)$ greater than the flow [33]. Thus, any graph and an associated TM can be represented by a unique point in the region bounded by these limits.

While this distinction is well-established [33], we strengthen the point by showing that *it can lead to incorrect decisions when evaluating networks.* Specifically, we will exhibit a pair of graphs $A$ and $B$ such that, as shown in Figure 2.2, $A$ has higher throughput but $B$ has

[1] We pick one for simplicity, and sparsest cut has an advantage in robustness. Bisection bandwidth is forced to cut the network in equal halves, so it can miss more constrained bottlenecks that cut a smaller fraction of the network.

higher sparsest cut. If sparsest cut is the metric used to choose a network, graph $B$ will be wrongly assessed to have better performance than graph $A$, while in fact it has a factor $\Omega(\sqrt{\log n})$ worse performance!

**Graph A** is a clustered random graph adapted from previous work [34] with $n$ nodes and degree $2d$. $A$ is composed of two equal-sized clusters with $n/2$ nodes each. Each node in a cluster is connected by degree $\alpha$ to nodes inside its cluster, and degree $\beta$ to nodes in the other cluster, such that $\alpha + \beta = 2d$. $A$ is sampled uniformly at random from the space of all graphs satisfying these constraints. We can pick $\alpha$ and $\beta$ such that $\beta = \Theta(\frac{\alpha}{\log n})$. Then, as per [34] (Lemma 3), the throughput of $A$ (denoted $T_A$) and its sparsest cut (denoted $\Phi_A$) are both $\Theta(\frac{1}{n \log n})$.

Let graph $G$ be any $2d$-regular expander on $N = \frac{n}{dp}$ nodes, where $d$ is a constant and $p$ is a parameter we shall adjust later. **Graph B** is constructed by replacing each edge of $G$ with a path of length $p$. It is easy to see that $B$ has $n$ nodes. We prove in Appendix A, the following theorem.

**Theorem 2.1:** $T_B = O(\frac{1}{np\log n})$ and $\Phi_B = \Omega(\frac{1}{np})$

In the above, setting $p = 1$ corresponds to the 'expanders' point in Figure 2.2: both $A$ and $B$ have the same throughput (within constant factors), but the $B$'s cut is larger by $O(\log n)$. Increasing $p$ creates an asymptotic separation in both the cut and the throughput such that $\Phi_A < \Phi_B$, while $T_A > T_B$.

**Intuition.** The reason that throughput may be smaller than sparsest cut is that in addition to being limited by bottlenecks, the network is limited by the total volume of "work" it has to accomplish within its total link capacity. That is, if the TM has equal-weight flows,

$$Throughput\, per\, flow \leq \frac{Total\, link\, capacity}{\#\, of\, flows\, \cdot\, Avg\, path\, length}$$

where the total capacity is $\sum_{(i,j)\in E} c(i,j)$ and the average path length is computed over the flows in the TM. This "volumetric" upper bound may be tighter than a cut-based bound.

## 2.2 TOWARDS A WORST-CASE THROUGHPUT METRIC

Having exhausted cut-based metrics, we return to the original metric of throughput. We can evaluate network topologies directly in terms of throughput (via LP optimization software) for specific TMs. The key, of course, is how to choose the TM. Our evaluation can include a variety of synthetic and real-world TMs, but we also want to evaluate topologies' robustness to *unexpected* TMs.

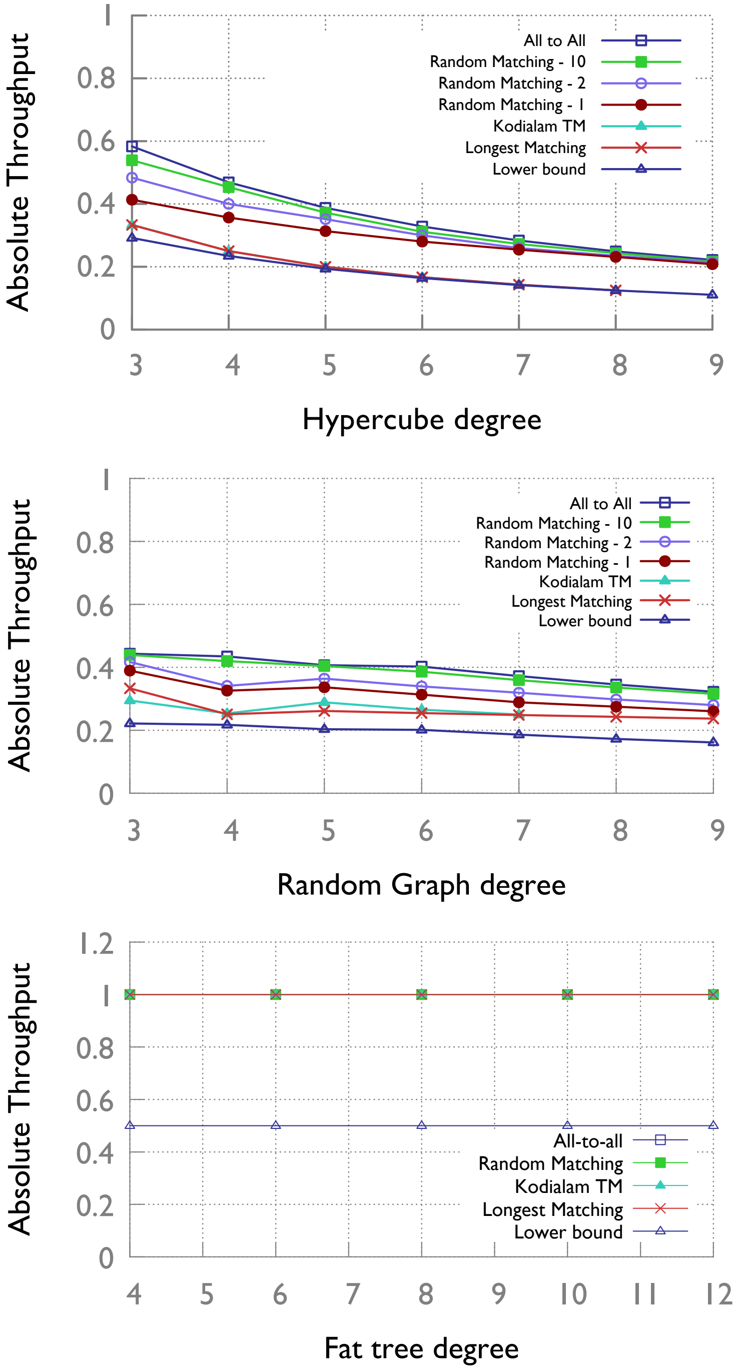


Figure 2.3: Throughput resulting from different traffic matrices in three topologies: hypercube, random graphs and fat trees

If we can find a worst-case TM, this would fulfill our goal. However, computing a *worst-case TM* is an unsolved, computationally non-trivial problem [35].[2] Here, we offer an efficient heuristic to find a *near-worst-case TM* which can be used to benchmark topologies.

We begin with the complete or **all-to-all TM** $T_{A2A}$ which for all $v, w$ has $T_{A2A}(v, w) = \frac{1}{n}$. We observe that $T_{A2A}$ is within 2× of the worst case TM. This fact is simple and known to some researchers, but at the same time, we have not seen it in the literature, so we give the statement here and proof in Appendix B.

**Theorem 2.2:** Let $G$ be any graph. If $T_{A2A}$ is feasible in $G$ with throughput $t$, then any hose model traffic matrix is feasible in $G$ with throughput $\geq t/2$.

Can we get closer to the worst case TM? In our experience, TMs with a smaller number of "elephant" flows are more difficult to route than TMs with a large number of small flows, like $T_{A2A}$. This suggests a **random matching TM** in which we have only one outgoing flow and one incoming flow per server, chosen uniform-randomly among servers.

Can we get *even closer* to the worst-case TM? Intuitively, the all-to-all and random matching TMs will tend to find sparse cuts, but only have average-length paths. Drawing on the intuition that throughput decreases with average flow path length, we seek to produce traffic matrices that force the use of long paths. To do this, given a network $G$, we compute all-pairs shortest paths and create a complete bipartite graph $H$, whose nodes represent all sources and destinations in $G$, and for which the weight of edge $v \rightarrow w$ is the length of the shortest $v \rightarrow w$ path in $G$. We then find the maximum weight matching in $H$. The resulting matching corresponds to the pairing of servers which maximizes average flow path length, assuming flow is routed on shortest paths between each pair. We call this a **longest matching TM**, and it will serve as our heuristic for a near-worst-case traffic.

Kodialam et al. [36] proposed another heuristic to find a near-worst-case TM: maximizing the average path length of a flow. This **Kodialam TM** is similar to the longest matching but may have many flows attached to each source and destination. This TM was used in [36] to evaluate oblivious routing algorithms, but there was no evaluation of how close it is to the worst case, so our evaluation here is new.

Figure 2.3 shows the resulting throughput of these TMs in three topologies: hypercubes, random regular graphs, and fat trees [15]. In all cases, A2A traffic has the highest throughput; throughput decreases or does not change as we move to a random matching TM with 10 servers per switch, and progressively decreases as the number of servers per switch is decreased to 1 under random matching, and finally to the Kodialam TM and the longest

[2] Our problem corresponds to the separation problem of the minimum-cost robust network design in [35]. This problem is shown to be hard for the single-source hose model. However, the complexity is unknown for the hose model with multiple sources which is the scenario we consider.

matching TM. We also plot the lower bound given by Theorem 2.2: $T_{A2A}/2$. Comparison across topologies is not relevant here since the topologies are not built with the same "equipment" (node degree, number of servers, etc.)

We chose these three topologies to illustrate cases when our approach is most helpful, somewhat helpful, and least helpful at finding near-worst-case TMs. In the **hypercube**, the longest matching TM is extremely close to the worst-case performance. To see why, note that the longest paths have length $d$ in a $d$-dimensional hypercube, twice as long as the mean path length; and the hypercube has $n \cdot d$ uni-directional links. The total flow in the network will thus be $(\# \, flows \cdot average\, flow\, path\, length) = n \cdot d$. Thus, all links will be perfectly utilized if the flow can be routed, which empirically it is. In the **random graph**, there is less variation in end-to-end path length, but across our experiments the longest matching is always within 1.5× of the provable lower bound (and may be even closer to the true lower bound, since Theorem 2.2 may not be tight). In the **fat tree**, which is here built as a three-level non-blocking topology, there is essentially no variation in path length since asymptotically nearly all paths go three hops up to the core switches and three hops down to the destination. Here, our TMs are no worse than all-to-all, and the simple $T_{A2A}/2$ lower bound is off by a factor of 2 from the true worst case (which is throughput of 1 as this is a non-blocking topology).

The longest matching and Kodialam TMs are identical in hypercubes and fat trees. On random graphs, they yield slightly different TMs, with longest matching yielding marginally better results at larger sizes. In addition, longest matching has a significant practical advantage: it produces far fewer end-to-end flows than the Kodialam TM. Since the memory requirements of computing throughput of a given TM (via the multicommodity flow LP) depends on the number of flows, longest matching requires less memory and compute time. For example, in random graphs on a 32 GB machine using the Gurobi optimization package, the Kodialam TM can be computed up to 128 nodes while the longest matching scales to 1,024, while being computed roughly 6× faster. Hence, we choose longest matching as our representative near-worst-case traffic matrix.

Contrary to our findings, Yuan et al. concludes in [37] that fat trees perform better than or similar to Jellyfish under HPC workloads. We evaluate throughput with paths restrictions as proposed under the LLSKR scheme in [37] and find that their conclusions are flawed. We further observe that the erroneous results stem from the inaccuracy in throughput computation. The value of flow on a sub-path in [37] is determined by the bottleneck link. This naive approach yields a feasible flow, but not a maximum flow in the network. The maximum flow may have larger flow values on some sub-paths compared to others. Note that this property of maximum flow (unequal distribution of flows across

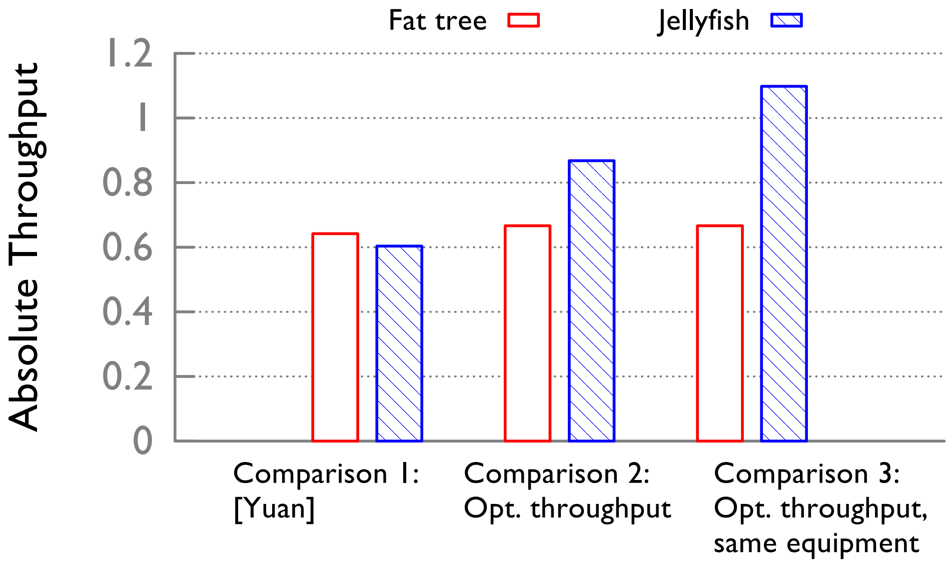


Figure 2.4: Naive throughput vs Actual throughput

different subpaths) holds irrespective of the objective - be it improving the minimum or average. Hence, maximum flow need to be computed as a solution to the LP associated with multi-commodity flow problem irrespective of the objective. Another issue with the comparison in [37] is the larger number of servers added to Jellyfish compared to Fat tree. In Figure 2.4, we observe that fat tree with 80 switches and 128 servers and corresponding Random Regular Graph (RRG) used in the paper with 80 switches and 160 servers haw similar throughput under naive throughput computation. However, when the number of servers are reduced to 128, the performance improves by 20%. However, the throughput is 38% higher in RRG compared to fat trees when the number of servers are the same and the maximum flow is computed.

### 2.2.1 Summary and Implications

Directly evaluating throughput with particular TMs using LP optimization is both more accurate and more tractable than cut-based metrics. In choosing a TM to evaluate, both “average case” and near-worst-case TMs are reasonable choices. Our evaluation will employ multiple synthetic TMs and measurements from real-world applications. For near-worst-case traffic, we developed a practical heuristic, the longest matching TM, that often succeeds in substantially worsening the TM compared with A2A.

Note that measuring throughput directly (rather than via cuts) is not in itself a novel idea: numerous papers have evaluated particular topologies on particular TMs. Our contribution is to provide a rigorous analysis of why cuts do not always predict throughput; a way to generate a near-worst-case TM for any given topology; and an experimental evaluation benchmarking a large number of proposed topologies on a variety of TMs.

## 2.3 EVALUATION

In this section, we present our experimental methodology and detailed analysis of our framework. We attempt to answer the following questions: Are cut-metrics indeed worse predictors of performance? When measuring throughput directly, how close do we come to worst-case traffic?

### 2.3.1 Methodology

Before delving into the experiments, we explain the methods used for computing throughput. We also elaborate on the traffic workloads and topologies used in the experiments.

**Computing throughput:** Throughput is computed as a solution to a linear program whose objective is to maximize the minimum flow across all flow demands. We use the Gurobi [38] linear program solver. Throughput depends on the traffic workload provided to the network.

**Traffic workload:** We evaluate two main categories of workloads: (a) real-world measured TMs from Facebook clusters and (b) synthetic TMs, which can be uniform weight or non-uniform weight. Synthetic workloads belong to three main families: all-to-all, random matching and longest matching (near-worst-case). In addition, we need to specify where the traffic endpoints (sources and destinations, i.e., servers) are attached. In structured networks with restrictions on server-locations (fat-tree, BCube, DCell), servers are added at the locations prescribed by the models. For example, in fat-trees, servers are attached only to the lowest layer. For all other networks, we add servers to each switch. Note that our traffic matrices effectively encode switch-to-switch traffic, so the particular *number* of servers doesn't matter.

**Topologies:** Our evaluation uses 10 families of computer networks. Topology families evaluated are: BCube [17], DCell [19], Dragonfly [39], Fat Tree [40], Flattened butterfly [41], Hypercubes [42], HyperX [43], Jellyfish [23], Long Hop [44] and Slim Fly [45]. For evaluating the cut-based metrics in a wider variety of environments, we consider 66 non-computer networks – food webs, social networks, and more [46].

### 2.3.2 Do cuts predict worst-case throughput?

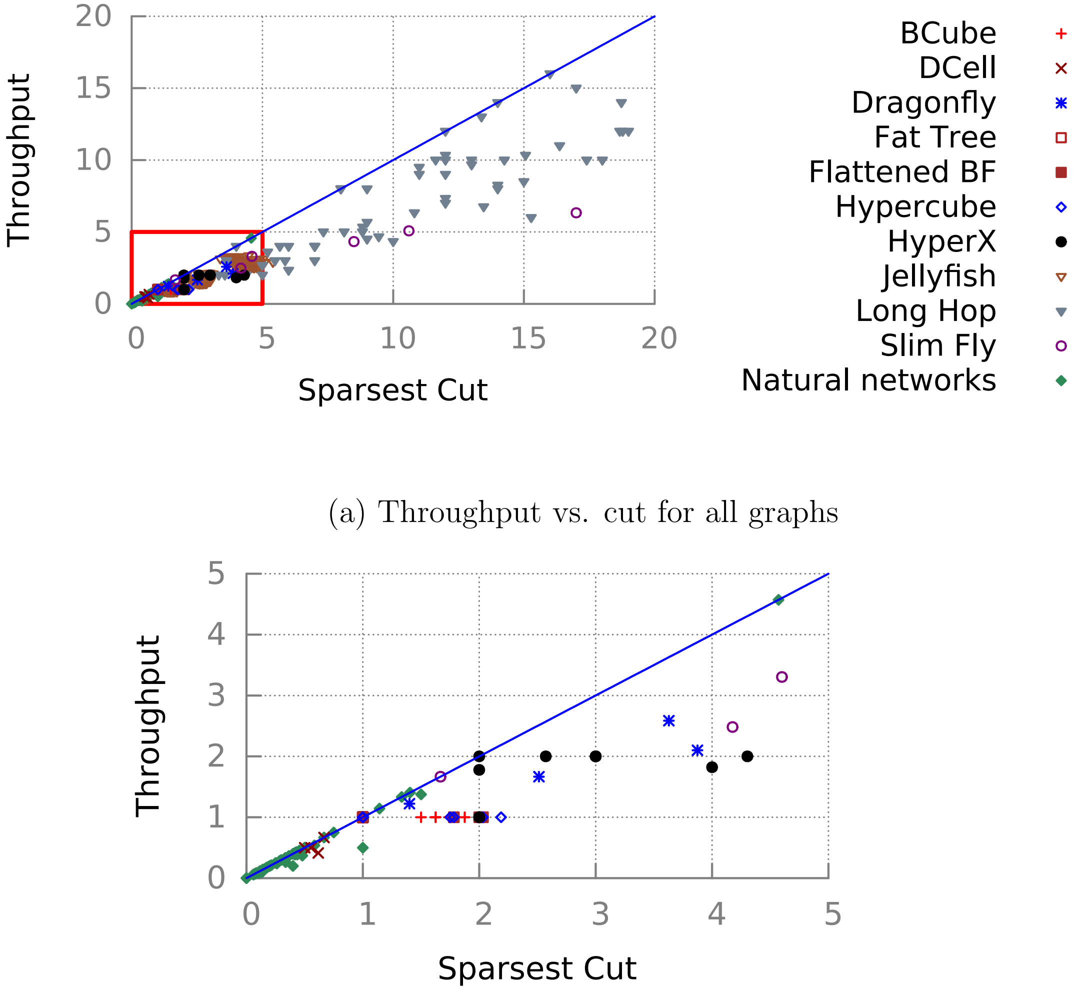
In this section, we experimentally evaluate whether cut-based metrics predict throughput. We generate multiple networks from each of our topology families (with varying parameters such as size and node degree), compute throughput with the longest matching TM, and find sparse cuts using heuristics with the same longest matching TM. We show that:

- In several networks, bisection bandwidth cannot predict throughput accurately. For a majority of large networks, our best estimate of sparsest-cut differs from the computed worst-case throughput by a large factor.
- Even in a well-structured network of small size (where brute force is feasible), sparsest-cut can be higher than worst-case throughput.

Since brute-force computation of cuts is possible only on small networks (up to 20 nodes), we evaluate bisection bandwidth and sparsest cut on networks of feasible size (115 networks total – 100 Jellyfish networks and 15 networks from 7 families of topologies). Of the 8 topology families tested, we found that bisection bandwidth accurately predicted throughput in only 5 of the families while sparsest cut gives the correct value in 7. The average error (difference between cut and throughput) is 7.6% for bisection bandwidth and 6.2% for sparsest cut (in networks where they differ from throughput). Maximum error observed was 56.3% for bisection bandwidth and 6.2% for sparsest cut.

Although sparsest cut does a better job at estimating throughput at smaller sizes, we have found that in a 5-ary 3-stage flattened butterfly with only 25 switches and 125 servers, the throughput is less than the sparsest cut (and the bisection bandwidth). Specifically, the absolute throughput in the network is 0.565 whereas the sparsest-cut is 0.6. This shows that even in small networks, throughput can be different from the worst-case cut. While the differences are not large in the small networks where brute force computation is feasible, note that since cuts and flows are separated by an $\Theta(\log n)$ factor, we should expect them to grow.

Sparsest cut being the more accurate cut metric, we extend the sparsest cut estimation to larger networks. Here we have to use heuristics to compute sparsest cut, but we compute all of an extensive set of heuristics (limited brute-force, selective brute-force involving one or two nodes in a subset, eigenvector-based optimization, etc.) and use the term **sparse cut** to refer to the sparsest cut that was found by any of the heuristics. Sparse cuts differ from throughput substantially, with up to a 3× discrepancy as shown in Figure 2.5. In only a small number of cases, the cut equals throughput. The difference between cut and throughput is pronounced. For example, Figure 2.5(b) shows that although HyperX networks of different



(a) Throughput vs. cut for all graphs

(b) Throughput vs. cut for selected graphs (zoomed version of (a))

Figure 2.5: Throughput vs. cut. (Comparison is valid for individual networks, not *across* networks, since they have different numbers of nodes and degree distributions.)

sizes have approximately same flow value (y axis), they differ widely in sparsest cut (x axis). This shows that estimation of worst-case throughput performance of networks using cut-based metrics can lead to erroneous results.

### 2.3.3 Does longest matching approach the true worst case?

We compare representative samples from each family of topology under four types of TM: all to all (A2A), random matching with 5 servers per switch, random matching with 1 server per switch, and longest matching. Figure 2.6 shows the throughput values normalized so that

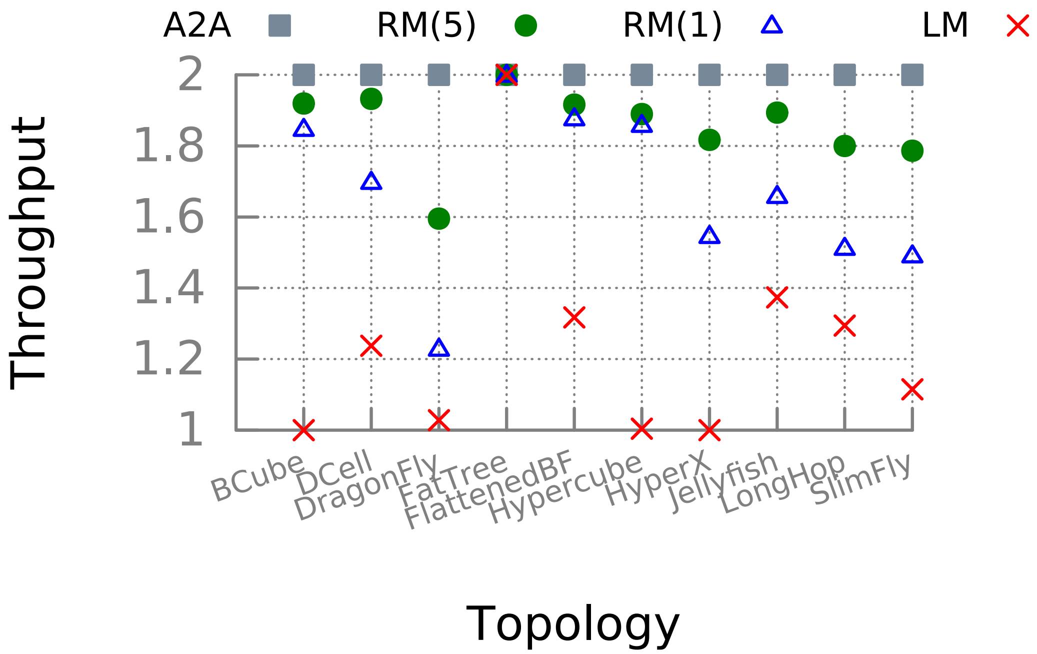


Figure 2.6: Comparison of throughput under different traffic matrices normalized with respect to theoretical lower bound

the theoretical lower bound on throughput is 1, and therefore A2A's throughput is 2. For all networks, $T_{A2A} \geq T_{RM(5)} \geq T_{RM(1)} \geq T_{LM} \geq 1$, i.e., all-to-all is consistently the easiest TM in this set, followed by random matchings, longest matching, and the theoretical lower bound. (As in Figure 2.5, throughput comparisons are valid across TMs for a particular network, not across networks since the number of links and servers varies across networks.)

Our longest matching TM is successful in matching the lower bound for BCube, Hypercube, HyperX, and (nearly) Dragonfly. In all other families except fat trees, the traffic under longest matching is significantly closer to the lower bound than with the other TMs. In fat trees, throughput under A2A and longest matching are equal. This is not a shortcoming of the metric, rather it's the lower bound which is loose here: in fat trees, it can be easily verified that the flow out of each top-of-rack switch is the same under all symmetric TMs (i.e., with equal-weight flows and the same number of flows into and out of each top-of-rack switch).

In short, these results show that (1) the heuristic for near-worst-case traffic, the longest matching TM, is a significantly more difficult TM than A2A and RM and often approaches the lower bound; and (2) throughput measurement using longest matching is a more accurate estimate of worst-case throughput performance than cut-based approximations, in addition to being substantially easier to compute.

| Topology family | All-To-All | Random Matching | Longest matching |
|---|---|---|---|
| BCube (2-ary) | 73% | 90% | 51% |
| DCell (5-ary) | 93% | 97% | 79% |
| Dragonfly | 95% | 76% | 72% |
| Fat tree | 65% | 73% | 89% |
| Flattened BF (2-ary) | 59% | 71% | 47% |
| Hypercube | 72% | 84% | 51% |

Table 2.1: Relative throughput at the largest size tested under different TMs

### 2.3.4 Topology evaluation

In this section, we present the results of our topology evaluation with synthetic and real-world workloads, and our near-worst-case TM.

But first, there is one more piece of the puzzle to allow comparison of networks. Networks may be built with different equipment – with a wide variation in number of switches and links. The raw throughput value does not account for this difference in hardware resources, and most proposed topologies can only be built with particular discrete numbers of servers, switches, and links, which inconveniently do not match.

Fortunately, uniform-random graphs as in [23] can be constructed for any size and degree distribution. Hence, random graphs serve as a convenient benchmark for easy comparison of network topologies. Our high-level approach to evaluating a network is to: (i) compute the network's throughput; (ii) build a random graph with precisely the same equipment, i.e., the same number of nodes each with the same number of links as the corresponding node in the original graph, (iii) compute the throughput of this same-equipment random graph under the same TM; (iv) normalize the network's throughput with the random graph's throughput for comparison against other networks. This normalized value is referred to as **relative throughput**. Unless otherwise stated, each data-point is an average across 10 iterations, and all error bars are 95% two-sided confidence intervals. Minimal variations lead to narrow error bars in networks of size greater than 100.

**Synthetic Workloads:** We evaluate the performance of 10 types of computer networks under uniform-weight synthetic workloads. Performance non-uniform weight synthetic workloads can be found in [47].

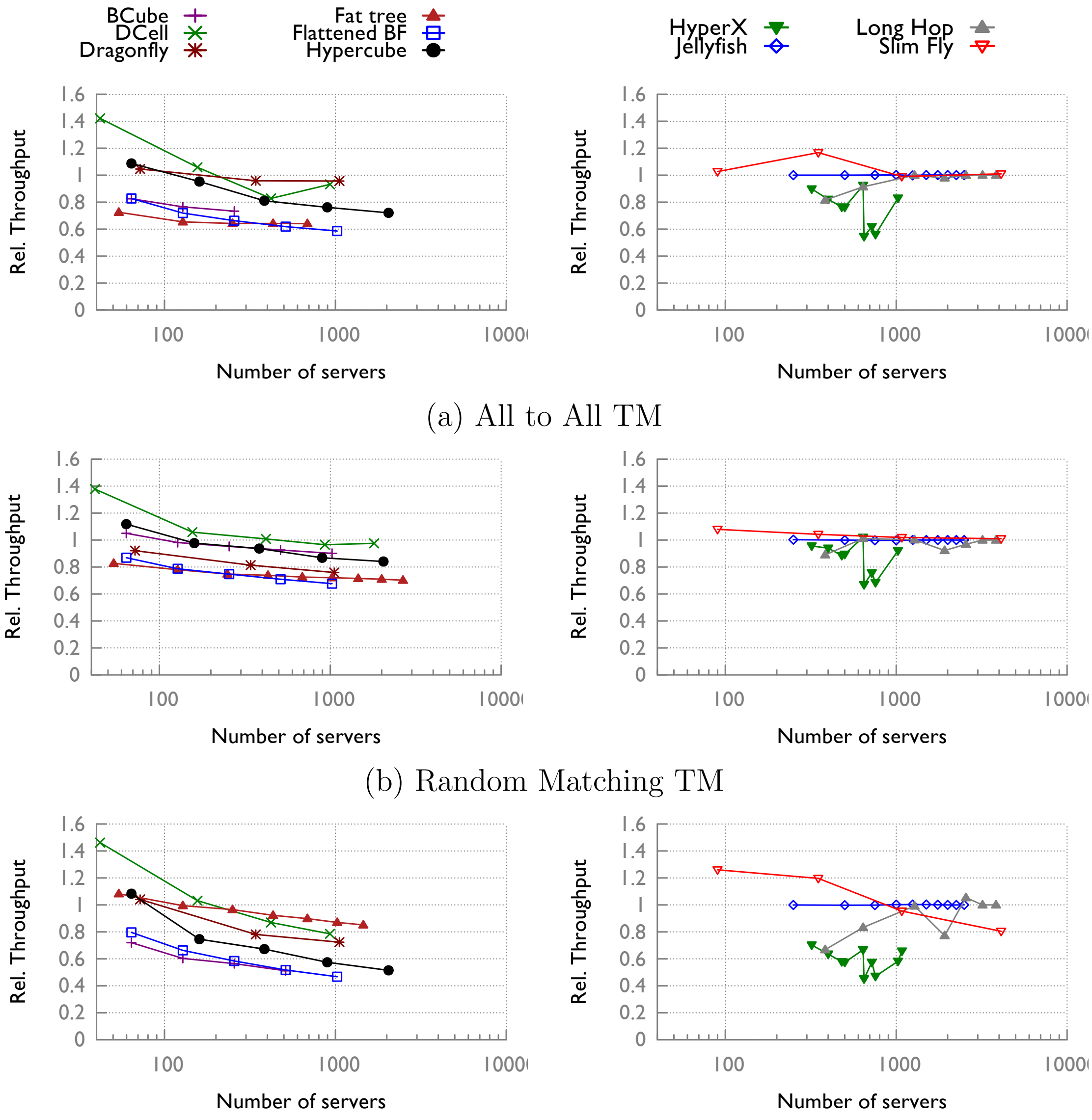
We evaluate three traffic matrices with equal weight across flows: all to all, random matching with one server, and longest matching. Figures 2.7 shows the results, divided among two figures for visual clarity. Most topologies are in the left, while the figures on the right show Jellyfish, Slim Fly, Long Hop, and HyperX.

**Overall,** performance varies substantially, by around 1.6× with A2A traffic and more than 2× with longest matching. Furthermore, for the networks of Figure 2.7, which topology "wins" depends on the TM. For example, Dragonfly has high relative throughput under A2A but under the longest matching TM, fat trees achieve the highest throughput at the largest scale. However, in all TMs, Jellyfish achieves highest performance (relative throughput = 1 by definition), with longest matching at the largest scale (1000+ servers). In comparison with random graphs, performance degrades for most networks with increasing size. The relative performance of the largest network tested in each family in the first set in Figure 2.7 is given in Table 2.1. HyperX [43] has irregular performance across scale due to choices of underlying topology. Given a switch radix, number of servers and desired bisection bandwidth, HyperX attempts to find the least cost topology constructed from the building blocks – hypercube and flattened butterfly

**Real-world workloads:** Roy et al. [1] presents relative traffic demands measured during a period of 24 hours in two 64-rack clusters operated by Facebook. The first TM corresponds to a Hadoop cluster and has nearly equal weights. The second TM from a frontend cluster is more non-uniform, with relatively heavy traffic at the cache servers and lighter traffic at the web servers. Since the raw data is not publicly available, we processed the paper's plot images to retrieve the approximate weights for inter-rack traffic demand with an accuracy of $10^i$ in the interval $[10^i, 10^{i+1})$ (from data presented in color-coded log scale). Since our throughput computation rescales the TM anyway, relative weights are sufficient. Given the scale factor, the real throughput can be obtained by multiplying the computed throughput with the scale factor.

We test our slate of topologies with the Hadoop cluster TM (TM-H) and the frontend cluster TM (TM-F) and the results are presented in Figures 2.8 and 2.9 respectively. When a topology family does not have a candidate with 64 ToRs, the TM is downsampled to the nearest valid size (denoted as Sampled points in the plot).

We also consider modifying these TMs by shuffling the order of ToRs in the TM (denoted as Shuffled points in the plot). Under the nearly uniform TM-H, shuffling does not affect throughput performance significantly (Figure 2.8). However, under TM-F with skewed weights, shuffling can give significant improvement in performance. In all networks except Jellyfish, Long Hop, Slim Fly and fat trees, randomizing the flows, which in turn spreads the large flows across the network, can improve the performance significantly when the traffic matrix is non-uniform. Note that relative performance here may not match that in Figure 2.7 since the networks have different numbers of servers and even ToRs (due to downsampling required for several topologies). However, we observe that the relative performance between



(a) All to All TM

(b) Random Matching TM

(c) Longest Matching TM

Figure 2.7: Comparison of TMs on topologies

topologies remains consistent across both the traffic matrices.

### Summary

In this chapter, we have experimentally shown that:

- Cuts (as estimated using the best of a variety of heuristics) differ significantly from worst-case throughput in the majority of networks.
- Our longest matching TM approaches the theoretical lower bound in all topology families. They are significantly more difficult to route than all-to-all and random

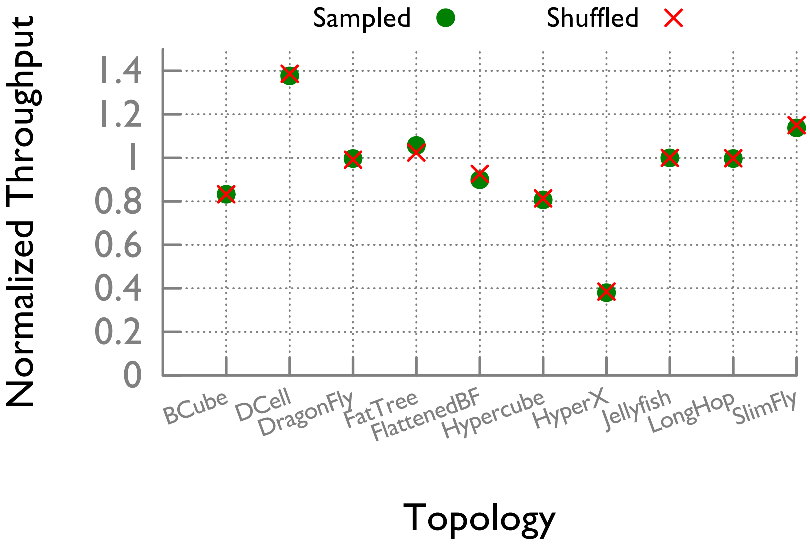


Figure 2.8: Comparison of topologies with TM-H( [1])

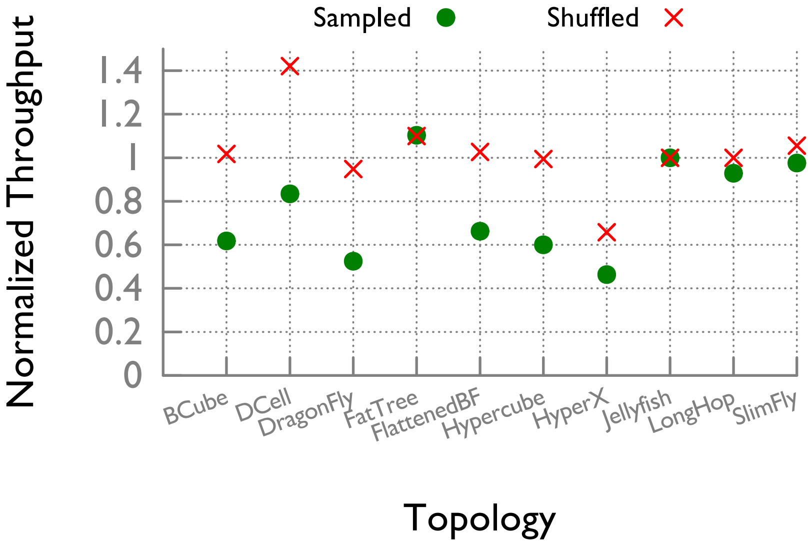


Figure 2.9: Comparison of topologies with TM-F( [1])

matching TMs.

- At large sizes, Jellyfish, Long Hop and Slim Fly networks provide the best throughput performance.
- Designing high throughput networks based on proxies for throughput may not yield the expected results (demonstrated with bisection bandwidth in HyperX).
- All networks except fat trees have graceful degradation in throughput under non-

uniform TMs when the fraction of large flows increases. Poor performance of fat trees is due to heavy load at the ToRs.

- Randomizing the placement of heavy flows could improve the performance for throughput-intensive applications.

## Chapter 3: SOURCE-ROUTED DATA CENTER FABRIC

Data center network architecture is moving towards a network fabric abstraction with the core of the network only providing simple forwarding functionality and complex functions delegated to the edge. This decoupling of forwarding from other network functions provides us with the opportunity to rethink the design of fabric towards a more flexible and efficient forwarding core [48]. How does the fabric achieve high performance? The key is traffic engineering: the network controller or distributed agents must select paths and load-balance between them to adapt to dynamic traffic patterns. This is a difficult technical problem due to the scale, dynamics, and high performance requirements of modern data centers. Since modifying network-wide switch forwarding entries is slow [49], the most common approach is two-stage: first a set of paths are proactively encoded into the data plane, and then the "edge" of the network load-balances among these in real time. Here the edge may be a server, a hypervisor, or a top-of-rack (ToR) switch; and the edge may optimize its path choice autonomously or as instructed by a central controller.

Encoding the paths into the data plane turns out to be non-trivial. A key constraint is the limited capacity of switch forwarding tables, ranging from roughly 2,000 (in some commodity OpenFlow gear) to 200,000 (for very simple exact-match MAC tables). To encode a diverse set of paths between each source-destination pair, a range of designs have proliferated [50, 51, 52, 53, 54, 18, 55, 16, 56]. These designs are (variously) topology-dependent, utilize large numbers of forwarding table rules, complex, and yet still limit the selection of paths. For example, CONGA [50] allows a sender to specify the path only up to a spine or top-level switch. Planck [51] and Shadow MACs [57] severely limits throughput for certain traffic patterns and topologies. XPath [53] performs a complex compression to encode a large number of paths; it can use more than 100k forwarding rules and still is not guaranteed to provide all paths.

Our goal is to make the case that the fabric should provide a simple abstraction: The ability to use any physical path. This enables *flexibility* in the sense that any available resource can be used without constraints. Furthermore, we argue that the *right architecture to achieve a flexible fabric is source routing:* the sender at the edge specifies a switch-level path through the network, and switches simply match on identifiers referring to their outgoing ports. In a sense, this approach takes the fabric design to its purest form: the switch is stripped down to the minimal functionality necessary for flexible high performance.

The most obvious benefit of this design is tiny forwarding tables (so small that it opens the possibility of simpler switch hardware). But, as we will argue later, a source-routed fabric is a broader architectural win: it improves achievable throughput, is more robust to connectivity failures, localizes switch forwarding table configurations so they are not dependent on network-wide topology, and essentially eliminates the need for careful updates to forwarding state in fabric switches [49]. It also may improve monitoring and security via path provenance, and ease data plane verification.

OpenFlow does not support IPv4 (loose or strict) source routing, and this would anyway lead to large headers. Another method is to use a stack of MPLS labels, 4 bytes per hop with pushing, swapping, and popping along the path. We show an even more compact method is possible: using arbitrary bit masks in OpenFlow 1.3, paths of sufficient length can be encoded in a single header field (with around 8-10 bits per hop). Perhaps surprisingly, we can even avoid any packet header modification in the switching fabric.

We are not the first to suggest source routing techniques for the data center [58, 54]. In contrast to that work, our contributions are to: (1) make the case for source routing as an *effective architecture for a flexible fabric*, (2) describe a compact method for implementing source routing within OpenFlow, and (3) quantify the improvements in forwarding table size and achievable throughput and how they depend on the traffic matrix.

## 3.1 RELATED WORK ON DATA CENTER CONTROL SCHEMES

Several traffic engineering (TE) techniques have been proposed to improve the performance of data centers. In this section, we elaborate on the state-of-the-art data center forwarding techniques and their limitations.

The majority of the schemes have been tailored to Clos networks, commonly used in data centers. In the discussion, we will refer to two families of topology. A **leaf-spine** topology is a two-tier Clos network. The switches in the lower tier are called leaves and those in the upper tier are called spines or core nodes. Each leaf is connected to every spine. A **fat tree** is a multi-tier Clos network. In the construction of [15], which we assume throughout the rest of the paper, a fat tree is a three-tier network which, when built using switches of $k$ ports, has $\frac{5}{4}k^2$ switches and up to $\frac{k^3}{4}$ hosts.

**CONGA** [50] is a traffic engineering scheme designed for leaf-spine topologies. Each leaf keeps track of congestion on its paths and chooses the least-congested path for a newly arriving flowlet. Tags in the VXLAN header are used to identify the next-hop spine node as well as to carry congestion information. CONGA relies on source routing at the first hop to allow the destination to keep track of the path used. However, spine nodes rely on the destination leaf address in the packet for forwarding and hence, requires a forwarding table which can accommodate the number of leaves in the network. The technique was evaluated on two-level Clos networks in [50], and as the network grows to multiple levels it would be increasingly difficult to encode all paths as the number of possible paths would grow exponentially.

**Shadow MACs** [57] is a forwarding scheme for traffic engineering which uses multiple spanning trees rooted at each destination. Each spanning tree has an associated "shadow" MAC address which allows shifting between the trees through MAC rewriting. Thus, for each destination address, only a limited number of paths are available. For example, **Planck** [51] uses a shadow MAC scheme with four spanning trees per destination. In addition to the overhead in MAC address rewriting, the limited path availability can impact throughput performance.

**XPath** [53] relies on compression to fit a large number of paths into switches' forwarding tables. It uses a two-step process involving aggregation of paths into path sets and assignment of IDs to each path set. Several optimizations reduce the computation for structured networks, but computation time for random networks is still very high (several hours to days). Hence, this is a computationally intensive and complex technique for achieving explicit path control in data centers. [59] also attempts to assign IDs to paths to facilitate concise representation. MAC addresses are assigned as IDs to paths such that multiple paths with that share a link can be aggregated.

A few data center TE schemes have used or proposed the use of source routing before. **FastPass** [52] is a novel traffic engineering scheme where a centralized controller decides the path as well as the time slot of transmission for each packet in the network. This TE scheme is restricted to tiered networks which are rearrangeably non-blocking. The successful implementation of this scheme would require an efficient source routing scheme. The paper suggests the use of VLANs, IP-in-IP tunnelling or ECMP spoofing for the implementation of source routing. In **SlickFlow** [54], packet headers contain a source route for a primary path and an alternate path. Since the packet contains next hop information for both paths, packet headers are larger. Moreover, the technique requires changes in the core of the network to handle the new SlickFlow header. **SecondNet** [58] is a data center virtualization architecture which uses port-switching based source routing (PSSR) as the forwarding method, implemented using MPLS. MPLS is one of the implementation options we discuss later.

Although source routing has been proposed for data center networks in the above papers, an analysis of impact of source routing has been absent. While [60] makes an attempt in this direction, the objective of the design was to reduce the controller load and optimize its placement. Our paper contributes a performance analysis of the impact of source routing, proposes compact implementations in OpenFlow, and makes the broad architectural case for source routing as a fabric substrate.

## 3.2 THE CASE FOR A SOURCE ROUTED FABRIC

In this section, we discuss why a source-routed fabric is an architectural win over a traditional IP-based forwarding fabric.

**Smaller forwarding tables**: In basic source routing, each switch only needs to store one rule for each outgoing port. For ease of implementation, we will propose schemes which need slightly more: one rule per (hop number, outgoing port) pair. But even in that case, the forwarding table grows linearly with the diameter of the network and is otherwise independent of the number of switches. Thus, source routing can support *all* possible paths to all destinations in the network while reducing the number of forwarding entries by several orders of magnitude. In the future, this may enable simpler and cheaper switch hardware.

**Higher throughput:** Unlike IP-based forwarding fabric which encounters path constraints due to forwarding table size restrictions, source routing can support any valid path in the network. This allows us to utilize the full path diversity of the network to achieve higher throughput.

**Nearly-static forwarding tables:** In source routing, a forwarding table entry at a node is updated only during addition or deletion of a link directly connected to it. Thus, forwarding table entries have only local dependencies. Hence, forwarding tables are unaffected by failures or addition of nodes in the broader network, leading to reduced error and complexity. In contrast, traditional forwarding has global dependenices.

**Faster response to failures:** Source routing allows routing via any available path. The edge-based reaction to failure can happen more quickly than global routing protocol reconvergence or reaction by a central controller.

**Architectural solution for consistent update:** Since source routing supports all feasible paths in the network, switching paths is just a matter of changing a header field at the sender. This essentially solves the consistent update problem of IP forwarding fabrics [61, 49] where special care is needed because switching paths requires state changes at multiple switches in the network.

**Ease of verification:** Formal data plane verification [62, 63, 64, 65, 66] of network policies is an important aspect of operation in modern data center networks. For IP-based forwarding, continuous modeling of all rules on all switches are needed because they run distributed routing protocols that change over time. However, with source routing, the fabric's packet forwarding is stable and predictable. Hence, verification of encapsulation at edge routers is sufficient.

## 3.3 POSSIBLE IMPLEMENTATIONS

We propose several possible implementations of source routing that are feasible in current data centers. We do assume a centralized controller which has a global view of the network and is capable of sending necessary information to the edge (edge router or hypervisor). The implementation involves two major stages—how routes are selected at the edge, and the encoding of the source route. We enumerate multiple techniques for each stage, which have varying implications for switch hardware and traffic engineering scheme.

**Source route encoding** In source routing, the entire path is inserted in the packet header. We need a mechanism that allows each intermediate router to read its next hop from the encoding. Depending on the capabilities of switches in the network, one of the following methods can be chosen for source routing in the core of the network. The methods are listed in increasing order of packet overhead.

**(a) Bit mask and TTL:** OpenFlow 1.3 allows arbitrary bit masks which can match any bits in a given field. These bits need not be adjacent or at the beginning of the field. This feature can be used to reduce the overhead associated with path information in the packet header. If the maximum degree (number of ports) of any switch in the network is 256, 8 bits are sufficient to uniquely identify a next hop. In such a network, a 32-bit field can hold path information for 4-hop paths. If the largest switch has 64 ports, using a single IPv6 address field, we could accommodate 21-hop paths ($\lfloor 128/6 \rfloor$). This allows a very compact representation of the route in the packet header.

At each switch, we need to identify the correct set of bits to be read. We propose to rely on an existing hidden pointer in the packet. Time-To-Live (TTL) is an 8-bit field in IP/MPLS headers that keeps track of the lifespan of a packet in the network. If we know the TTL set by the source, the number of hops traversed by the packet can be easily deduced. Thus, TTL can be used as a pointer which gives the location of the path corresponding to the current hop. As a simple example, if the TTL is set to 255 at the source, the maximum degree in the network is 16 (4 bits) and 3 hop paths are supported, a mask of xxxx1111xxxx can be used when the current TTL is 253. Note that this technique can be used only if every router in the network is guaranteed to perform TTL operations correctly.

**(b) Bit mask and pointer:** In networks where TTL functions are unavailable, an additional pointer field can be used to locate the correct next-hop information. The pointer field is initialized to 1 and incremented at each hop. In addition to bit-mask match rule, each packet has to be matched against an additional rule which increments the value of the pointer. This can be implemented in OpenFlow with appropriate matching on the combination of the pointer and route fields, with a rewrite action applied to the pointer.

**(c) Bit mask on switch IDs:** In contrast to all other schemes presented here, it is actually possible to implement source routing without any header field rewrites. To accomplish this, we specify the path as a list of globally-unique switch IDs. A path that passes through a specific switch contains its own ID followed by the next-hop neighbor's ID. Hence the forwarding table contains match rules for each pair of adjacent locations in the path with the first entry being the switch's own ID and the second entry as any of its neighbors' ID. This technique does not require a pointer to the current location because it is implicit in the position of a switch's own ID in the path. Unlike the previous schemes, the number of bits required per tag is not dependent on the largest degree, but on the number of switches in the network.

**(d) MPLS-based source routing:** In networks that do not support OpenFlow 1.3 features, Multi Protocol Label Switching [67] can be used instead of compact representation and arbitrary bit masks. However, the default stack size of MPLS labels in most routers is three. Although it can be increased to four or five, this also increases header overhead since each label requires 4 bytes, so MPLS is more appropriate in networks with small diameter.

**Route selection at the edge** In a data center with virtual machines running client applications, security concerns preclude insertion of source routes at the client VM. Hence, the route can be injected either at the hypervisor or at the edge router.

**(a) Hypervisor:** The hypervisor can be programmed to encapsulate packets received from the attached VMs with the appropriate source route as well as decapsulate packets received from the network. The relevant source route can either be directly obtained from the centralized controller or computed at the hypervisor based on network conditions. The network state could be directly obtained from the controller or could be learned through any distributed algorithm that runs across hypervisors in the network. A suitable path computation technique can be adopted depending on the traffic engineering scheme.

**(b) Edge router:** The source routes may also be inserted at the edge routers in the network. The path computation needs to be done at the centralized controller, which pushes instructions to the edge routers. The routing table at edge routers will be much larger compared to the rest of the network, as they need to accommodate rules for source path header encapsulation and decapsulation.

Note that this proposal is essentially agnostic to the traffic engineering scheme. We expect that source routing will benefit traffic engineering schemes broadly, by providing more choice in paths.

## 3.4 EXPERIMENTAL ANALYSIS OF SOURCE-ROUTED FABRIC

In this section, we analyze the impact of source routing and several past traffic engineering schemes with respect to throughput performance. More results on the impact on forwarding table size can be found in [14].

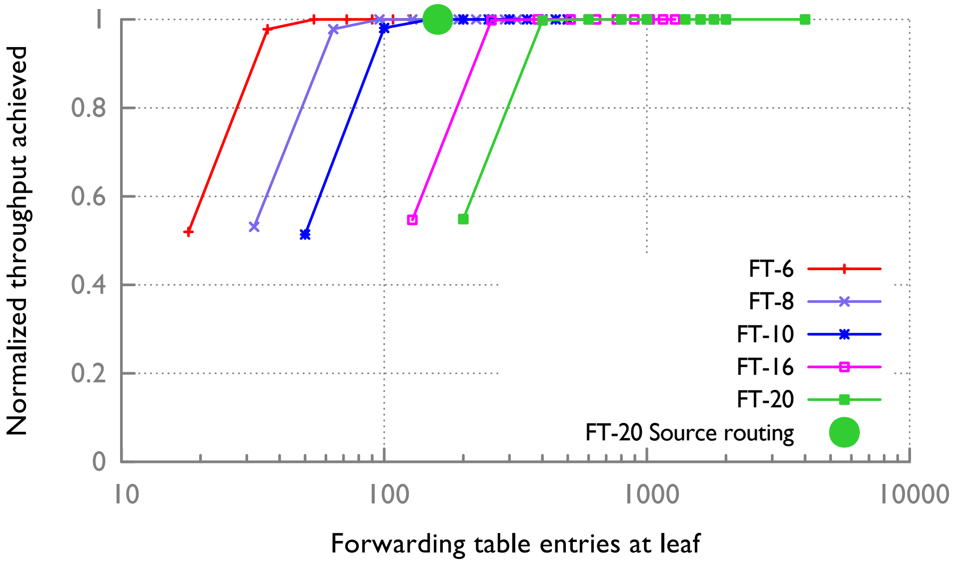


Figure 3.1: Fat trees A2A TM

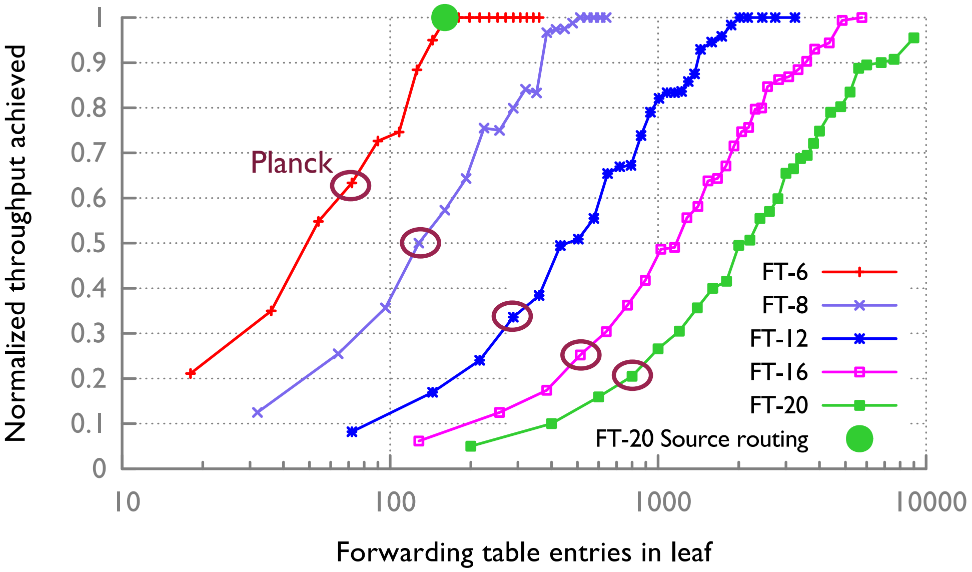


Figure 3.2: Fat trees Random Matching TM

### (A) Methodology

In this section, we present results on evaluation of various Traffic Matrices on the commonly used fat tree topology. More results can be found in [14].

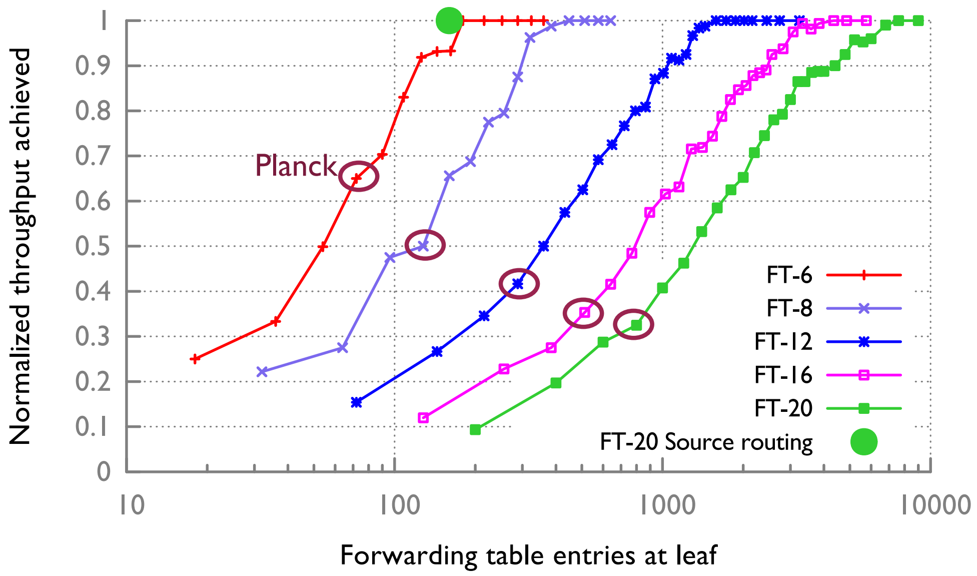


Figure 3.3: Fat trees Random Matching TM with 10% of flows with 10× demand

**Understanding the topologies: Leaf-spine** topology with $L$ leaves and $S$ spine switches has $L * (L-1)$ source-destination leaf pairs, each of which can use any of the $S$ (shortest) spine paths, leading to a total of $L * (L-1) * S$ paths. The spine switch only has to save information on each of the destination leaves leading to $L$ entries. However, each leaf switch has $(L-1)S$ entries for $S$ paths to each of the other leaves in the traditional shortest paths-based routing scheme.

In a **fat tree** constructed with degree $K$ switches, there are $K^2/2$ leaves in the lower layer. The number of leaf pairs is $K^4/4$. Connecting each pair of leaves, we can have $K^2/4$ paths (considering $K/2$ next-hops between each adjacent layers). Hence, the total number of valid paths in the network is $K^6/16$. At each leaf, we can have $K^2/2$ outgoing flows to other leaves each of which can use at most $K^2/4$ paths. Hence, the total number of forwarding table entries will be $K^4/8$.

**Traffic matrix for throughput comparison:** We use Topobench [6] to evaluate throughput performance of various schemes. Throughput is computed as a solution to a linear program whose objective is to maximize the minimum flow across all demands. We extend the framework to accommodate a constrained set of paths according to each forwarding scheme. We evaluate three realistic traffic matrices (TMs): (a) All-to-all (A2A) TM with a flow between every pair of switches, (b) Random Matching TM with one outgoing and one incoming flow per switch, chosen uniform-randomly and (c) Random matching with non-uniform traffic where the matching between servers is uniform-random, but 10% of the flows have a demand 10× bigger than the rest of the flows. A flow is defined per leaf pair

| Topology | Total paths | Forwarding table entries in a leaf switch | | | |
|---|---|---|---|---|---|
| | | Traditional | CONGA | PLANCK | Source routing |
| Leaf-spine ($L$ leaves, $S$ spines) | $L(L-1)S$ | $(L-1)S$ | $(L-1)S$ | $4(L-1)$ | $8S$ |
| Fat trees (degree $K$) | $K^6/16$ | $K^4/8$ | - | $2K^2$ | $8K$ |
| Leaf-spine (256 leaves, 32 spines) | 2088960 | 8160 | 8160 | 1020 | 256 |
| Fat trees (degree 20) | $4 * 10^6$ | $2 * 10^4$ | - | 800 | 160 |

Table 3.1: Forwarding table size comparison

and is considered to be the aggregate traffic between hosts in these leaves.

A traffic pattern similar to all-to-all is generated by certain real-world applications (such as the shuffle phase in MapReduce). Random matching is a more demanding traffic pattern [6] since it has a single large flow exiting each leaf. This is somewhat similar to a situation that could occur if an application is allocated all the machines in two racks and performs large transfers between the two halves of the application (e.g. a shuffle). Non-uniform random matching is a representative of the realistic scenario where a small fraction of flows dominate in size, alongside other smaller demands spread throughout the network.

We compare the performance of traditional routing (all shortest paths through a router saved in its routing table), CONGA, Planck and source routing. For source routing, we assume that IPv6 address is used to carry the source route in a network with switch degree at most 256. Hence, the source routing design used for evaluation supports paths of length up to 16.

### (B) Forwarding table size

Forwarding table size in switches is limited due to high cost and power consumption. A typical size of forwarding tables in high-end data center switches today is 144K entries [68], which may not be sufficient to accommodate all paths. The problem is aggravated in low end switches with forwarding table size less than 7K [69]. We compare the requirements of various data center forwarding schemes.

A leaf in the **leaf-spine** topology uses $L \cdot S$ entries under CONGA. Since Planck limits the number of paths to 4 for each destination, the number of forwarding table entries at each leaf is $4L$ in Planck. With source routing, we have a constant number of entries – $8S$. Although we have paths of length 16, a leaf can appear only at an odd position and a spine can appear only at an even position in the path. Hence each switch has to check only half of the locations. This property holds for all "level-based" networks with links only between adjacent levels. A switch at an odd level can appear only at an odd hop in a path.

In a **fat tree topology** with degree $K$, a leaf switch in Planck has $2K^2$ paths. On the other hand, source routing requires only $8K$ forwarding table entries to support paths up to length 16 in fat tree with shortest path length 5. CONGA does not scale to accommodate three level topologies. Note that it is possible to do hierarchical routing in fat trees with small forwarding tables if control over the chosen path is not required. However, this can lead to congestion and uneven utilization of the network. As we will see, for better utilization of the network and efficient traffic engineering, it is necessary to have information on all paths at the leaves.

A summary of the comparison is given in Table 3.1. The general trend in the growth of forwarding table size in fat trees with increase in degree is given in Figure 3.4. We can see that, without any optimizations, forwarding table requirements of traditional routing with all shortest paths cannot be accommodated even in high-end switches for fat trees with degree as small as 34. Planck, with a 4-path limit per destination, cannot support fat trees of degree more than 60, built from low-end switches with $7K$ routing table size. On the other hand, forwarding table requirements of source routing uses only a fraction of the memory in low-end switches. Also, note that the source routing scheme used here can support any paths of length up to 16. Traditional scheme and Planck require forwarding tables which are much larger, while only supporting shortest paths (as shown in the Figure).

**(C) Throughput Analysis**

Due to the limited forwarding table size, data center switches can accommodate only a limited number of paths at any point in time. This can affect the throughput performance of applications. In this section, we analyze the impact of reducing the number of available paths on maximum achievable throughput. Although several factors can contribute to reduction in throughput, particularly transport mechanisms, this evaluation focuses solely on the limitation imposed by constraints on the available forwarding paths.

We evaluate three realistic traffic matrices (TMs): (a) All-to-all (A2A) (b) Random matching and (c) Random matching with non-uniform traffic. Throughput is normalized with respect to the maximum throughput achievable in the topology using the same TM with no path constraints.

A fat tree built from degree $K$ switches is denoted by FT-K. Figure 3.1 shows the variation of normalized throughput in fat trees with all-to-all TM as a function of the number of forwarding table entries at each leaf switch. Under the dense (A2A) TM, 2-3 paths per flow

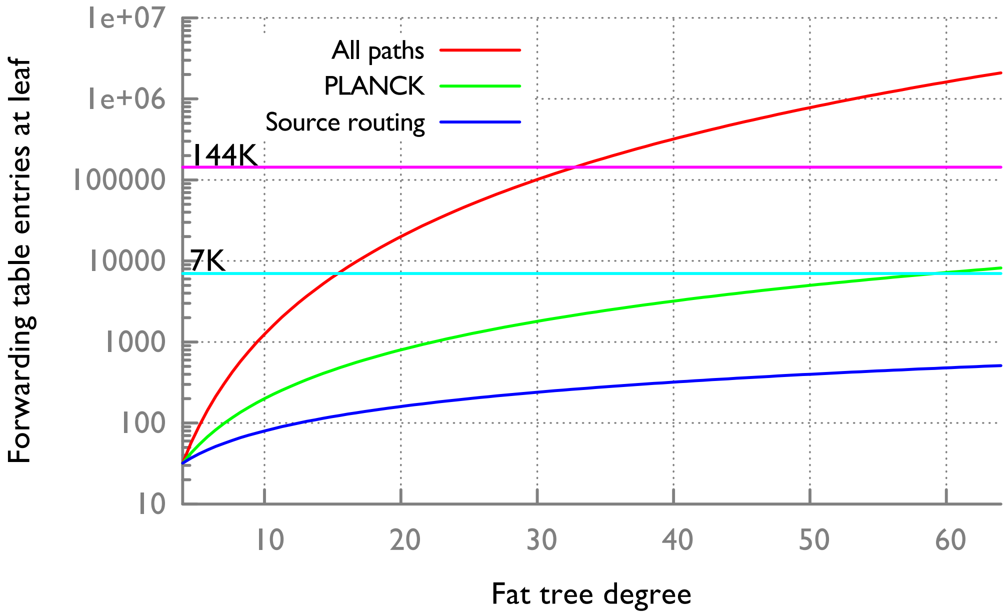


Figure 3.4: Forwarding table size for fat trees

is sufficient for good performance. However, the forwarding table entries grow exponentially with size of the fat tree network.

Figure 3.2 shows normalized throughput in fat trees under the uniform random matching TM. Limited paths are chosen randomly with additional constraints to use a diverse set of links. In order to minimize overlap between paths, two feasible next hops are picked randomly at each level and the one with fewer paths through it is chosen during the path assignment phase. We observe that the number of forwarding table entries required to maintain the same throughput performance increases dramatically compared to the A2A TM. We also note that performance of Planck with 4 paths per destination degrades as the swich degree (and network size) increases.

Normalized throughput in fat trees with random matching TM and 10% of the flows with 10× bandwidth demand is given in Figure 3.3. While each flow has equal priority with the uniform traffic matrix, large demand flows are more significant with non-uniform traffic matrix. With minimal overlap between the larger flows, the non-uniform TM can achieve maximal throughput with slightly fewer paths. However, the basic trend remains consistent across the uniform and non-uniform random matching TMs since the bottleneck under path constraints is the number of paths itself.

The main take-away point is: **A large number of paths between racks can be used effectively.** Generally, splitting a flow into a large number of components can affect its performance; $K^2/4$ paths per flow might seem hard to handle. However, we are considering aggregate rack-to-rack traffic which is composed of traffic from $(K/2)^2$ host pairs. When a large number of hosts contribute towards the aggregate flow, the number of paths assigned to each host-based flow can be limited. Hence, the paths made available by source routing can be effectively used to improve throughput. Source routing also allows non-shortest paths (useful in failure scenarios).

As data centers move towards network fabric-based architecture, there is an increasing need for a flexible and scalable routing scheme at the core. Under our preliminary investigation, source routing appears to be a technique that fits the bill. With source routing, the forwarding table can easily fit within the available memory of even low-end switches due to its linear dependence on node degree and network diameter. Moreover, throughput performance of the fabric will not be negatively impacted by limited availability in network paths, when source routing allows the edges to pick any feasible path in the network. Due to its flexibility and scalability, source routing is a suitable candidate for network fabric architecture.

In order to strengthen this routing scheme further, we need to tackle a few issues. First, this work has not dealt with response to switch and link failures. It is essential to build an auxiliary mechanism to route around failures, either with additional information in the header or through a network response mechanism. Second, in order to efficiently utilize the multitude of paths made available by source routing, it is essential to have an efficient traffic engineering scheme. Although a few of the existing schemes rely on several variations of source routing, they are tightly coupled with certain topologies and the performance is far from optimal. With source routing at the core of the design, it will be possible to use any valid path in any topology leading to greater flexibility in the design and implementation of traffic engineering schemes.

# Chapter 4: CONTROLLING THOUSANDS OF MICRO DATA CENTERS

Micro data centers (MDCs) at the edge are emerging as prominent components in the Internet infrastructure. Traditionally, MDCs were used for content caching and video performance optimization in Content Delivery Networks [70] and for user load-balancing at entry/exit of content provider networks such as Google [11], Facebook [10], and Microsoft [9]. With the emergence of novel applications with stringent performance requirements, the role of MDCs is expanding. MDCs providing limited compute resources very close to users are increasingly used for supporting a variety of low-latency applications, often augmenting, and sometimes replacing, the traditional hyperscale clouds.

An important use case for increased processing at the edge is the burgeoning Internet of Things (IoT). IoT devices produce data in the form of video, voice, sensor information, etc., which needs to be analyzed and acted upon, often within tight delay constraints. Processing and compressing data from such applications at the edge will also reduce the bandwidth demand in the core. Availability of compute in close proximity is a must-have for offloading Augmented and Virtual Reality (AR/VR) that operate at timescales comparable to the sensitivity of human perception. Self-driving and connected cars will also increasingly rely on these edge DCs [71, 72]. MDCs can also prove beneficial to other latency-sensitive applications with heavy compute requirements such as real-time video analytics and online gaming.

Today, carrier networks are spearheading large-scale deployments of MDCs for supporting Virtualized Network Functions (VNFs). Service providers like AT&T [73] and Verizon [74] are converting traditional Central Offices (COs) with dedicated hardware to MDCs with off-the-shelf servers. A single carrier operates a few thousand COs – AT&T has 4700 COs in the US [75] – and with the advent of 5G, the number of MDC sites is expected to grow further. In addition to the critical cellular services, these MDCs are also designed to support emerging applications like those above [72].

Thus, Wide Area Networks (WANs) are morphing from a network of routers to an interconnection between a multitude of geographically distributed micro data centers. We refer to this emerging model as a *WAN As a Network of Data centers (WAND)*. WAND is indispensable to providing support for the emerging hyperconnected world with billions of devices and geo-distributed applications.

In order to meet the goals of geo-distributed applications while simultaneously utilizing resources efficiently, we need an efficient WAND resource management scheme. However, this is difficult for several reasons. First, the combination of scale and geographic spread has not been addressed by prior large-scale systems (Table 4.1). Second, the environment needs to support a motley set of applications with diverse requirements. This includes long-running streaming applications (e.g., cellular VNFs, other middlebox service chains), batch analytics (e.g., cellular log analytics, Hadoop jobs) and Lambda-like short-lived jobs (e.g., elastic web servers). To meet the requirements of such geo-distributed high-performance applications, resource allocation on distributed MDCs and the interconnecting WAN will need to be coordinated. Third, the smaller size of MDCs and the potential for demand bursts mean that resource availability in any particular MDC will be more dynamic and variable than in a hyperscale DC. In other words, MDCs enjoy limited benefits of statistical multiplexing. Thus, WAND is characterized by its *scale*, *geographic spread*, *diversity of applications*, and *limited resources at MDCs*. While one or two of these challenges have been addressed in existing large-scale systems [7, 8, 9, 10, 11], the combination of all characteristics calls for novel resource management techniques in WAND.

We design an autonomous WAND control system, Patronus, which enables high utilization of the infrastructure while improving the performance of applications through (a) fast and efficient resource allocation, and (b) intelligent adaptation during variations. The centralized control plane of Patronus has several components. First, Patronus introduces a simple and expressive API which can support representation of diverse requirements of WAND applications using *WAND tags*. Tags encode bounds on application requirements such as latency, bandwidth, location preference, deadline, etc. Second, Patronus leverages predictability of traffic and resource usage patterns inherent to the WAND environment/applications to obtain a long-term perspective on resource availability and usage.

Third, Patronus provides fast and efficient resource allocation in the complex WAND environment through division of labor across an instantaneous scheduler and a long-term scheduler. The long-term scheduler relies on long-term predictions for packing of jobs across time while complying with deadlines, fairness, etc. The instantaneous scheduler allocates resources to a subset of tasks deemed active in the current instant by the long-term scheduler. Both schedulers rely on hierarchical optimization for handling diverse application objectives and a simple mechanism for converting *WAND tags* to constraints.

In this work, we make the following contributions:

- We identify control challenges in an emerging environment with geo-distributed MDCs and performance-sensitive applications, which we call WAND (WAN as a Network of Data Centers), and its differentiating features compared to other large-scale infrastructure.

- We design a simple and expressive WAND API for capturing the requirements of diverse WAND applications.

- We design Patronus, a dynamic WAND controller providing scalability and high utilization for the WAND provider and high performance for WAND user applications.
- Using realistic data, we evaluate trade-offs in performance and control overhead using closed-loop control in WAND to meet the high-level goals of applications and operators.
- We show that Patronus can schedule resources across thousands of MDCs in seconds while reducing peak resource usage by up to 47%.

## 4.1 RELATED DOMAINS

One may draw parallels between control of thousands of data centers and interconnecting network in WAND and several well-studied control domains such as clusters, traditional ISPs, private WANs, geo-distributed analytics, etc. Next, we argue why control solutions in these domains cannot be borrowed easily for WAND (overview in Table 4.1).

**Traditional ISPs:** Traffic engineering is a well-studied problem in traditional ISP networks with a single resource (geographically-distributed network). While the problem has been tackled from different perspectives ranging from oblivious routing [76, 77] to traffic-adaptive schemes [78, 79, 80, 81], these solutions do not apply to the WAND model. Extending network traffic classes and prioritization to an environment with demands and performance constraints on multiple resources (WAN network and DC resources like CPU, memory, etc.) is a non-trivial challenge. Besides, the presence of new applications over which the service provider has complete control (eg, background analytics jobs) will help the provider to drive the network and DCs to high utilization. This was difficult in traditional ISPs which mostly dealt with user-driven inelastic traffic. As ISPs move to SDN-based control, there are also efforts to scale SDN with distributed controllers. Recursive SDN [82] proposed a hierarchical solution for geo-distributed carrier networks. RSDN and other work [83, 84] in distributing SDN control, however, only considers network control.

**Private inter-data center WANs:** Inter-DC private WAN solutions such as B4 [7] and SWAN [8] optimize resources across multiple applications similar to WAND. However, private WANs with a sparse network connecting tens of hyperscale data centers operate at a much smaller scale (in terms of *number of distinct sites*) compared to the carrier environment composed of thousands of micro data centers and interconnecting links. Moreover, the proposed solutions only focus on the bandwidth requirements of the private WAN since the DCs are hyperscale and relatively less constrained (or at least have more reliable statistical multiplexing). In WAND, both DCs and the WAN are frequently constrained. Furthermore, quickly adapting to traffic variations is more critical in WAND due to the limited capacity of MDCs that cannot accommodate bursts.

**Cluster schedulers:** Cluster schedulers [85, 86, 87] are responsible for allocation of server resources within hyperscale DCs. They consider demands over multiple types of resources and strive to drive the cluster to high utilization, often with prioritization across applications. They deal with large scale, e.g., allocating at the machine or rack granularity across many racks. However, the resources are highly localized (typically within a single DC) in contrast with a geo-distributed WAND. Moreover, cluster schedulers are typically decoupled from the network controllers which manage symmetric high-bisection bandwidth intra-DC interconnect between servers. This separation of control may not work well in a WAND environment which has an irregular topology and expensive WAN links.

**Traffic-aware VM placement in data centers:** Traffic-aware VM placement is a related domain with joint server and network resource management within a DC. However, this problem is NP-hard, while the WAND resource management can be tackled using a Linear Program. In intra-DC environments, flexibility in VM placement translates to flexibility in the location of source and destination of the flow [88]. This optimization problem can be reduced to the Quadratic Assignment Problem (QAP) which is NP-hard. In the WAND model, the source and the destination of the flow are fixed (typically at the end-user). We have flexibility in choosing way-points, i.e., DCs. Thus, WAND resource management is a *resource-augmented* Multi-Commodity Flow (MCF) problem with location restrictions. The challenges in WAND are related to scalability and quick adaptation.

**Application-specific solutions:** There exist several solutions tailored to specific geo-distributed applications.

**(a)** ***NFV solutions*****:** Elastic Edge [89] is an application-agnostic framework which allocates resources across NFVs. However, this solution is restricted to scheduling within a single DC. PEPC [90] introduced a new architecture for the cellular core to solve state duplication across EPC components and improved throughput at bottlenecked VNFs. KLEIN [91], another cellular-specific solution, put forward a 3GPP-compliant solution for load-balancing cellular traffic across MDCs. However, this does not take into account the presence of multiple applications with diverse performance needs in the DCs, or the constraints on the interconnecting network between DCs.

**(b)** ***Geo-distributed analytics*****:** Solutions for big data analytics across the wide area [92, 93] offer optimization for low-latency processing of queries, but do not deal with the difficult problem of cross-application optimization.

**(c)** ***Multimedia Systems*****:** Multimedia delivery systems often employ resource optimization techniques spanning cloud to edge devices [94, 95, 96, 97, 98] to meet soft real-time requirements of this application. These techniques often focus on the resource-contrained mobile devices. MDCs in WAND offer greater flexibility.

**WAND:** Having established that existing resource management techniques cannot be directly extended to generic WAND, we give a brief overview of current domain-specific solutions offered by provider networks.

**(a)** ***Carriers*****:** AT&T [73] and Verizon [74] have published white papers on SDN/NFV reference architectures and are building platforms with similar features such as network and DC controllers, data monitoring, a policy module, an orchestration module, etc. In addition to cellular traffic, cellular MDCs also intend to support other micro-services and applications such as connected cars [72]. These platforms, however, are in the early stages of development and do not currently offer solutions for resource management at the scale of thousands of MDCs.

**(b)** ***Content Providers*****:** Points of Presence (PoPs) of large content providers like Google [11], Microsoft [9], and Facebook [10] also constitute a WAND. However, solutions in this domain primarily focus on load-balancing traffic across edge locations. Currently, they do not offer compute resources for other applications at the edge.

**(c)** ***CDNs*****:** Content Delivery Networks cache objects at the edge to serve user requests with minimal latency. CDNs are a sub-class of WAND where traffic is bandwidth-heavy, and edge resources are mostly used for storage and delivery. Like content providers, they do not typically support other user applications. Application-specific video delivery optimization schemes [70] have been proposed for this environment.

## 4.2 FEATURES OF WAND

Next, we crystallize the key characteristics of the WAND environment. In WAND, (i) WAN and DC components can be coordinated, (ii) MDCs are small to moderate in size but large in number, (iii) demand includes network and DC components, (iv) some traffic may be elastic, and (v) at individual MDCs, small scale means limited statistical multiplexing and high demand variability.

**A federated system:** A WAND includes both a WAN and MDCs, whose operation may be coordinated in a single system or may be composed of federated subsystems.

**Scale:** WAND comprises a network of a large number of data centers — typically hundreds to a few thousands — which are individually small to moderate in size. CDNs have tens

| | **Property** | **WAND** | **Traditional ISPs** | **Private Inter-DC WAN** | **Cluster Schedulers** | **Application-specific (e.g., geo-distributed analytics, NFV platforms, etc.)** |
|---|---|---|---|---|---|---|
| **Infrastructure** | **Geographically fragmented resources** | Yes | Yes | Yes | | Yes |
| | **Large scale** | Yes | Yes | | Yes | Yes |
| | **Cross-application optimization for high utilization** | Yes | | Yes | Yes | |
| **Applications** | **Demand over multiple resources** | Yes | | | Yes | Yes |
| | **Location constraints** | Yes | | | | Yes |

Table 4.1: Comparison of features — WAND and other infrastructure

to hundreds of locations. Carrier networks operate thousands of DCs. There are efforts to install micro data centers at every base station, which will increase the scale to tens of thousands [99].

**Demands with multiple resource requirements:** WAND applications use WAN bandwidth as well as multiple resources within the DC, including compute, memory, and bandwidth, among others. While this is true of applications outside of WAND as well, in a WAND these resources may be coordinated in a single system. In addition, the particular mix of resources needed is likely to evolve with new applications that employ MDCs: cellular control traffic adds load on NFVs at DCs, uses limited WAN throughput but is latency sensitive; an IoT data-aggregation application may have high throughput to the edge DC, high resource consumption in the DC and moderate throughput to remote hyperscale DCs. Applications may also be categorized as single-user and multi-user applications. Cellular VNFs are shared across users in a region. On the other hand, an enterprise VPN connection will be composed of dedicated VMs and tunnels for a single enterprise.

**Elasticity of traffic:** Depending on the types of applications running on WAND, some traffic may be elastic — a provider can control the rate and time of resource allocation in DCs and WAN based on current resource availability. This can be used for low priority traffic such as backups.

**Limited statistical multiplexing:** Edge DCs do not form a single large pool of compute. Each DC is small, and therefore subject to more load variability due to less statistical multiplexing. They are distributed geographically, hence they are non-interchangeable for latency-sensitive applications. This may also lead to more variable utilization on individual DCs during regional hotspots. Hence, efficient dynamic resource management is paramount.

We quantify the impact of limited statistical multiplexing in Figure 4.1. We use a real-world dataset from a DNS provider comprising of DNS queries from across the US over a single day tagged with origin zip code and timestamp (more details about the dataset in § 4.5.3 b). For measuring variability, we use the Coefficient of Variation, $CV$, defined as the ratio of standard deviation to the mean. $CV$ is a comparable measure for the extent of variability for distributions with different means.

We divide the data with samples every minute into chunks of different lengths (15 min and 60 min) and compute $CV$ over each chunk at different geographic scales — per edge location, per area[1] and over the complete dataset (entire country). In Figure 4.1, we plot the CDF of $CV$ over all chunks in one day across all edges/areas. We observe that $CV$ is the highest when locations are considered independently and the lowest in the complete dataset. When the timescale is increased from 15 min to 60 min, $CV$ remains nearly the same at larger scales while it increases more than 100× for DC-level traffic. This shows that edges are subjected to higher variability compared to traditional clouds and therefore, benefits accruable from statistical multiplexing are limited.

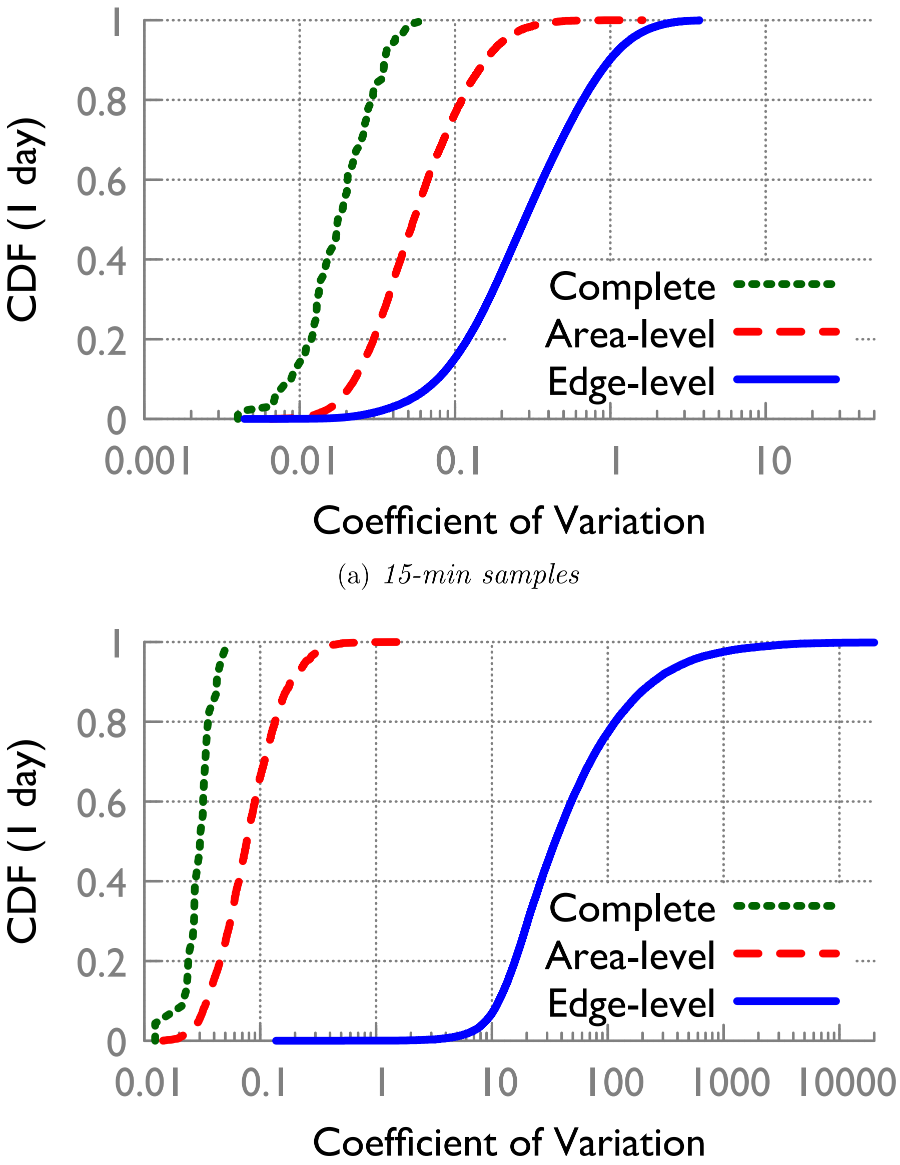
## 4.3 WAND API

Before delving into the design of a WAND API that helps applications specify their needs to the WAND controller, we discuss common requirements of WAND applications.

### 4.3.1 Application requirements

Geo-distributed applications in WAND can be of two general types: (a) streaming with real-time traffic or (b) batch analytics which processes offline data. Each has a variety of

[1] More details about area boundaries in § 4.5.3. Load in an area is the sum of loads across all edge locations in that area



(a) *15-min samples*

(b) *60-min samples*

Figure 4.1: Coefficient of Variation (CV) of traffic load based on a real-world DNS dataset evaluated at different scales — MDC-level, area-level, and the entire country.

performance requirements; we describe examples in order to understand how to build an expressive WAND API.

**Latency:** When an application prefers an MDC at the edge over a distant hyperscale cloud with cheaper resources, the main motivating factor is often latency. For user-facing applications, the latency constraint is primarily related to end-to-end latency experienced by users. Non-user facing applications may have latency requirements between different

| **Operation** | **Description** |
|---|---|
| **create** () | Returns *app_id* |
| **setType** (*app_id*, *app_type*) | *app_type* $\in$ $\{$*streaming, batch*$\}$ |
| **setUserGroups** (*app_id*, list[ ($user\_group\_id_i$, list[ $<tag_i>$) ]] ) | $user\_group\_id_i$ may be IP prefix, tags are optional |
| **setTasks** (*app_id*, list [ ($task\_id_j$, list[ $<tag_j>$ ] ) ] ) | each task may have one or more optional tags |
| **setDependencies** (*app_id*, list[ ($task\_id_m$, $task\_id_n$ , list[$<tag_{mn}>$ ]) ] ) | each dependency may have one or more optional tags |
| **setConstraints** (*app_id*, list[ ($tag_i$, *constraint_name*, *constraint_type*, *constraint_value*) ] ) | *constraint_name* $\in$ $\{$ *containers, latency, bandwidth, affinity, deadline*$\}$<br>*constraint_type* $\in$ $\{$ *min, max, sum*$\}$<br>For *latency*, value is time in milliseconds<br>For *bandwidth*, value is bandwidth in Mbps<br>For *affinity*, value is 0 for anti-affinity, 1 for affinity<br>For *deadline*, value is UNIX epoch time |
| **UpdateConstraints** (*app_id*, list[ ($tag_i$, *constraint_name*, *constraint_type*, *constraint_change*) ] ) | *constraint_type* $\in$ $\{$ *latency, bandwidth* $\}$ and Signed integer indicating change in latency/bandwidth |
| **setRedirectionOptions** (*app_id*, *redirect*) | *redirect* $\in$ $\{$*drop, edge, remote*$\}$ |
| **setApplicationPreferences** (*app_id*, *preference, value*) | Extendable list of preference attributes. Current set:<br>*preference* $\in$ $\{$*eviction_tolerance, optional_modules*$\}$<br>For *eviction_tolerance*, value $\in (0, 1]$<br>For *optional_modules*: value is list[ ($tag_j$, $weight_j$)] |
| **setMonitorInterval** (*app_id*, *interval*) | monitoring interval in milliseconds |

Table 4.2: WAND API fields

concurrent modules.

**Bandwidth:** An application running in an MDC may have bandwidth requirements to users or between different internal components.

**Deadlines:** Deadlines are primarily associated with batch processing jobs. They denote the final time before which the application expects an output. In applications with buffering capability, this can also be the maximum time to fill the buffer. For example, in video feeds to be processed the edge, the cameras may have limited storage capacity. In such cases, when the analytics is not time-constrained, the video may be pulled by the edge MDC with some time flexibility.

**Intra-application dependencies:** In traditional batch processing jobs, the application is represented using a Directed Acyclic Graph (DAG) where the nodes are tasks and the edges denote dependencies. The same notation may be used in WAND. However, in distributed analytics, we may have additional constraints based on location of data and latency/bandwidth constraints. Geo-distributed applications of streaming nature (NFV service chains, traditional stream processing, etc.) can also be represented using a graph. The key difference between streaming and batch jobs is that in streaming all components need to be active at the same time, whereas in batch jobs the tasks are executed sequentially in an order determined by dependencies.

In addition to task-level dependencies, WAND applications may also have state-based dependencies. For example, cellular customers directed to their home MDC does not incur control overhead related with state transfer, whereas those redirected to an MDC farther away will cause mobility-related overheads.

**External dependencies:** An application may have external dependencies on other jobs, type of resources, etc. For example, an analytics job on cellular traffic will prefer to be colocated with the cellular service chain. In addition to such affinity constraints, an application may have anti-affinity constraints (two customers may prefer to not have their VPN provider edges colocated).

**Evictions:** One of the fundamental characteristics a WAND has to deal with is limited statistical multiplexing due to the small size of MDCs. This manifests as high load variability and for medium- or low-priority applications, a risk of being evicted by a burst of load on a high-priority application. Applications may have different preferences regarding this risk, in at least two ways.

First, applications can differ in their tolerance for eviction, relative to the value they place on having resources close to users. An opportunistic application may prefer to grab resources at the edge whenever possible, if evictions are not especially costly (e.g., an opportunistic web cache at the edge can serve requests at lower latency when nearby MDC resources are available). On the other hand, a more conservative application might prefer to run in an MDC only when the probability of eviction is low (e.g., a real-time application serving self-driving cars).

Second, an application may wish to lessen the damage due to eviction by indicating which of its constituent modules are less critical. A video analytics application which processes CCTV feeds may have different levels of resource configuration [100] or a critical minimal set of cameras which provide high coverage. Specifying such intra-application criticality can increase the controller's scheduling flexibility and thus increase the application's chances of being scheduled.

**Grouping requirements:** We define a set of users which share the same characteristics, and therefore have a common set of requirements (latency, bandwidth, etc.), as a ***user group***. For example, in cellular environment, the set of users assigned to a specific HSS may be one user group. Such grouping makes it more convenient for applications to specify their needs.

### 4.3.2 Defining a WAND API

We devise an API that allows geo-distributed WAND applications to represent their diverse requirements. The key operations supported by the API are given in Table 4.2.

Both streaming and batch applications are represented using a DAG. In streaming the dependencies denote the connections between tasks running concurrently. In batch applications, the dependencies denote the sequence of execution. A tag may be associated with a node (task) or an edge (dependency) in the application graph. An application can use these tags to denote its preferences such as bandwidth and latency requirements. A tag may be of the type *min*, *max* or *sum*. When a tag specifies a *min* value, it is the minimum requirement on that attribute required by the application (e.g., minimum bandwidth). When a tag is specified as *sum*, the limit applies to sum of the attribute across all resources with the same tag. For example, the end-to-end latency requirement of a service chain can be specified using this type, with no constraints on individual dependencies.

The app_id also acts as the tag for the entire application. Hence, to represent external dependencies, an application may use the app_id of the external application in the SetConstraints() function. A specific task of an external application can be represented using the format app_id:tag. In order to represent location constraints, the infrastructure provider can provide location tags, either at per-MDC level or at a regional-level.

Our WAND API also supports an extendable list of application preferences. Current supported preferences are **eviction_tolerance**, which specifies an approximate acceptable probability of eviction of a resource allocated to the application; and **optional_modules**, which specifies constituent tasks within the application that it prefers to be evicted first, rather than evicting the entire application.

Currently, Patronus supports linear constraints only since the schedulers rely on specialized Linear Programs. However, this is sufficient to handle the requirements of most WAND applications we considered.

## 4.4 WAND CONTROL PLANE

We design an automated control system, Patronus, to achieve high efficiency in the WAND environment. Patronus is a centralized controller managing resources across distributed MDCs and the interconnecting WAN. Patronus is designed for a private WAND infrastructure, such as that operated by cellular providers, with a variety of streaming and batch applications over which the infrastructure provider has some control and visibility. We do not currently consider external workloads, and hence adversarial traffic. Automated control in Patronus is realized with a sense-control-actuate loop composed of three main components: the prediction module, the resource allocation module, and the monitoring module.

### 4.4.1 Prediction Module

The prediction module is responsible for accurately estimating the requirements of applications. Depending on the information made available by applications, the prediction may involve several phases. For all applications, the prediction module estimates the resource needs for future instances based on history. The prediction module is also responsible for estimating user traffic patterns, such as diurnal behavior, across time to facilitate long-term resource planning. For blackbox applications where the dependencies and requirements are not explicitly known, the prediction module also estimates these dependencies between modules.

### 4.4.2 Resource Allocation Module

The resource allocation module is responsible for efficiently allocating resources across WAND applications, both streaming and batch jobs with a wide range of requirements. The scheduling involves two phases: long-term scheduling and instantaneous scheduling. The long-term scheduler maintains a long-term plan of the entire WAND system. It is responsible for handling complex requirements such as deadlines, location constraints, fairness, etc. At the beginning of each scheduling interval, the long-term scheduler shares the subset of tasks and the predicted loads to the instantaneous scheduler for immediate scheduling. The unallocated tasks are returned to the long-term scheduler for future scheduling or offloading to remote data centers. This separation allows Patronus to have fast and simple instantaneous scheduling, while conforming to complex application requirements across longer timescales.

The pre-processing phase in resource allocation involves the conversion of application details obtained from the API to a concise set of actionable requirements easily parsable by the schedulers. The *min*, *max* and *sum* values associated with tags provide bounds on latency and bandwidth for tasks/dependencies. Latency bounds also determine a subset of DCs where the task may be scheduled.

Before we delve into the schedulers, we discuss hierarchical optimization and the conversion of application requirements to constraints which form the core of both long-term and instantaneous schedulers.

### Hierarchical optimization

Patronus employs incremental *multi-objective optimization* for scheduling diverse applications with a wide range of requirements. Unlike traditional Linear Programming (LP) with a single objective, this technique supports multiple objectives within the same optimization. This allows the WAND controller to allocate resources of different types across diverse applications in a single optimization with priorities on applications/resources/locations determined by the operator.

***Objectives:*** In the multi-objective optimization, each objective has four tunable parameters: priority ($P$), weight ($W$), absolute tolerance ($\alpha$) and relative tolerance ($\rho$). The LP solver repeats the optimization $n$ times in the order of priorities, where $n$ is the number of priority classes. When multiple objectives belong to the same priority class, the weight denotes the relative weight of each objective within the class. In this case, the aggregate objective of the priority class with multiple sub-objectives is given by the weighted sum. The absolute tolerance of an objective represents the absolute value of degradation in the optimal value tolerated while optimizing lower priority classes. The relative tolerance, $\rho \in [0, \inf]$, denotes the degradation tolerable as a multiplicative factor of the optimal value.

A simple example with two objectives on DC utilization for a single application with a single task — nearest MDC assignment and load balancing at DCs is given below. The highest priority objective (higher P value) minimizes the fraction of traffic allocated to each DC weighted by the distance from each user-group to the MDC. This is the nearest-MDC objective. Allowing 50% tolerance on this optimal value, the next round of optimization minimizes $K$, the difference in utilization between all pairs of MDCs. In short, this optimization supports nearest-MDC assignment with load-balancing across MDCs up to a certain limit of deviation from nearest MDC distance. For this example, the LP is:

| **Symbol** | **Description** |
|---|---|
| $M$ | The set of MDCs |
| $E$ | The set of WAN links |
| $C_m$ | Capacity of MDC $m$ (#containers) |
| $B_l$ | Capacity of link $l$ (Gbps) |
| $N_k^a$ | Number of containers consumed by task $k$ of app $a$ per unit traffic |
| $G^a$ | The set of user-groups, $g$, of app $a$ |
| $D_m^{a,g}$ | Distance from user-group $g$ of app $a$ to MDC $m$ |
| $u_g^a(t)$ | Traffic at $t$ for user-group $g$ of app $a$ |
| $f_m^{a,g}(t)$ | Traffic from user-group $g$ of app $a$ assigned to MDC $m$ at $t$ |

Table 4.3: ILP Notation

Obj0: P=2, W=1, $\alpha$=0, $\rho$=0.5

$$\text{Minimize} \quad \sum_{m \in M, g \in G^a} D_m^{a,g} * f_m^{a,g} \tag{4.1}$$

Obj1: P=1, W=1, $\alpha$=0, $\rho$=0

$$\text{Minimize} \quad K \tag{4.2}$$

Subject to:

$$\forall m \in M: \quad N_1^a * f_m^{a,g} \leq C_m \tag{4.3}$$

$$\forall m_1, m_2 \in M: \quad N_1^a * f_{m_1}^{a,g} - N_1^a * f_{m_2}^{a,g} \leq K \tag{4.4}$$

$$\forall g \in G^a: \quad \sum_{m \in M} f_m^{a,g} = u_g^a \tag{4.5}$$

Thus, hierarchical optimization allows combining multiple objectives within and across applications in Patronus with varying levels of tolerance determined by the operator. This technique forms the core of both long-term and instantaneous schedulers.

Long-Term Scheduler

Long-term scheduling in WAND offers two benefits: (i) Analysis of historical traffic patterns provides the long-term scheduler with an approximate estimate of resource usage far ahead in time. Most user-facing applications in WAND have diurnal traffic patterns with predictable patterns across days. (ii) For deadline-bound jobs, it allows planning resource allocation ahead of time to meet the deadline. The WAND environment may host a large number of opportunistic and flexible jobs which are not time-critical. Analyzing the complete set of such jobs and their extent of flexibility during instantaneous resource optimization will increase the complexity of the scheduler considerably. In order to speed up instantaneous allocation and ensure fairness/cost-sharing across time, scheduling of flexible jobs are handled through long-term resource planning.

In this phase, we also mitigate the impact of limited statistical multiplexing at individual MDCs. While each individual MDC in WAND suffers from high variability due to its small size, a pool of MDCs in a region enjoys less variability when aggregated resources are considered (as shown in Figure 4.1). In addition to identifying the most suitable MDC for a task, the long-term planner also identifies areas for backup scheduling. This loose allocation of tasks to of a group of MDCs during the long-term planning phase also helps avoid the need for a global search for an alternative allocation during instantaneous scheduling when the primary choice MDC is unavailable. Thus, long-term planning reduces the number of active low-priority tasks and their possible locations to be considered during a given instant while ensuring that the deadlines and other constraints are met in longer timescales.

The long-term scheduler can operate in two modes: (i) fast mode, (ii) replan mode. When a new job arrives at the long-term scheduler, first, the fast mode is initiated. The aggregate load at all DCs for all previously allocated jobs is used to determine the available capacity and an attempt is made to place the job. If its requirements (latency/bandwidth/deadline etc.) are met with allocation at preferred MDCs, the allocation is finalized. If the fast mode fails, a replan is invoked over applications at or below the priority class of the new job. In this phase, the resources are shared fairly across multiple jobs belonging to the same class. If the new job is not completely accommodated at the MDCs, parts of it may be scheduled on a regional or remote clouds in this phase. The long-term scheduler may also invoke replan of the schedule if long-term resource predictions change significantly relative to prior estimates.

The long-term scheduler is also responsible for providing inputs to instantaneous scheduling at each scheduling interval. It provides a limited set of feasible locations for each task based on traffic estimates. It also includes a set of optional tasks to ensure work-conservation during under-estimation of traffic.

### Instantaneous Scheduler

The instantaneous resource allocation module is responsible for per-instance scheduling of application components determined to be scheduled by the long-term scheduler. This includes (a) a subset of batch tasks, (b) loads on active streaming applications, and (c) optional tasks which are not critical in the current instance but may be scheduled if resources are available. These application components may belong to different priority classes and may have different objectives. For highest priority applications guaranteed to be accommodated by an MDC (such as cellular traffic which is guaranteed to have a utilization below a low threshold, say 50%), optimization is not invoked unless there is a failure in the system.

The instantaneous scheduler also relies on hierarchical optimization, but typically it has fewer objectives than the long-term planning mode. This is possible because deadlines, fairness, and the complete set of preferences are not handled by the instantaneous scheduler (long-term scheduler shares a critical subset). Even within the same class, the long-term planner will include relative priorities that determine which task is to be eliminated if sufficient capacity is not available. Such tasks are determined by long-term fairness estimates.

### Other Optimization Features

We discuss two features in the context of WAND optimization: eviction tolerance of WAND applications and geo-distributed fairness objectives employed by WAND provider.

**Eviction Tolerance:** Lower priority applications in WAND will be evicted when load of applications in higher priority classes spikes high enough that the MDC becomes overloaded. Therefore, being allocated resources in MDCs near users comes with some risk. We allow applications to control this tradeoff through the *eviction tolerance* parameter in the WAND API. During resource allocation, this high-level application requirement is incorporated in the optimization based on statistical analysis.

The goal of the eviction tolerance, $\theta$, is to represent the maximum eviction probability per unit-time tolerable by the application ($\theta \in [0, 1]$). However, evictions are not perfectly predictable. When considering scheduling an application in an MDC, Patronus attempts to predict eviction probability based on three factors: (i) mean load (M) of applications with priority $\geq c$, (ii) variability of their load ($V$), and (iii) resource requirements ($R$) of app $a$. Mean and variability are computed over a period of length equal to the length of the task being scheduled. i.e., if a task is 5 min long, the mean and variability of historical data from past 5 min is estimated. Variability is measured as Coefficient of Variation.

The application with demand $R$ is evicted when the higher priority load exceeds $C - R$ where $C$ is the capacity of the MDC. Assuming that the variability of this load follows a normal distribution (with mean $M$ and coefficient of variation $V$), the probability of eviction at an instant is $P(L > C - R) = 1 - P(L < C - R) = 1 - \Phi(\frac{C-R-M}{MV})$, where $\Phi$ is the cumulative probability of a standard normal distribution, $X \sim N(0, 1)$, $\Phi(x) = P(X < x)$.

Note that all the terms in the probability expression are constants that can be determined apriori based on historical data. An MDC is considered a candidate for placement only if this estimated probability of eviction for the app is smaller than the application's eviction tolerance ($\theta$). This is an approximate heuristic since the distribution of load may not be perfectly normal. However, the application can still adjust its eviction probability by increasing or decreasing $\theta$.

**Fairness:** The second optimization feature is the *fairness objective* for geo-distributed applications. The distributed nature of the applications and the underlying infrastructure calls for new notion of fairness. Defining fairness in WAND can be challenging for two main reasons: (i) WAND is a multi-resource environment with requirements on multiple resources, and (ii) in with most applications composed of geo-distributed components, it is difficult to define a domain of fairness. Does each application get a fair-share at each MDC? Can underallocation for an app in one MDC be compensated by overallocation at another location?

Patronus employs a simple notion of fairness. Since DCs are the bottleneck in current cellular WANDs, we use fair-share allocation across DC resources for applications. However, instead of fair-sharing at each individual DC, the fairness domain is *areas* (collective group of neighboring MDCs). The choice of area as the domain for determining fairness is justified by the behavior of applications in WAND and the structure of the underlying topology. Streaming applications such as cellular VNFs are capable of load-balancing across adjacent DCs. When a single edge is overloaded, the application can obtain its fair-share through a neighbor.

### 4.4.3 Monitoring Module

Monitoring of applications serves two purposes: (i) enable quick reaction to changes in application behavior and (ii) provide input for prediction and long-term planning. The monitor interval can be explicitly set by an application through the WAND API or determined by the provider during runtime based on the extent of variability. We develop a simple tunable monitoring module in Patronus which can run alongside applications in WAND. We assume that this monitored information is made available centrally to the resource allocation module. The prediction module that relies on the output of monitoring may be co-located in the same MDC or centralized.

## 4.5 EXPERIMENTS

### 4.5.1 Implementation

We implement Patronus as a stand-alone control system with pluggable modules for each component. The current implementation of various modules is described below.

**Resource allocation module:** Both the instantaneous and long-term schedulers are Java applications with an integrated Gurobi [38] environment for solving the Linear Programs associated with resource allocation. Both schedulers take as input application requirements as JSON files containing task-level/user-group level requirements and dependencies.

The long-term scheduler holds a plan which contains estimated allocations for all applications across time. We store a plan for 1 day in 1 minute time-slots. The instantaneous prediction runs every minute. These parameters are tunable. When a new application arrives in the system, it is initially placed in this plan by the long-term scheduler. At each allocation interval, the long-term scheduler sends the list of currently active tasks to the instantaneous scheduler for actual placement. The schedulers convert requirements to linear objectives and constraints in a hierarchical linear program and solve the multi-objective LP using Gurobi. The output from the instantaneous scheduler is the set of allocated tasks, their location and the amount of resources allocated. Currently, the schedulers consider resources at the scale of VMs/containers within data centers.

### 4.5.2 Evaluation

In this section, we evaluate scalability and efficiency of the Patronus system.

### 4.5.3 Methodology

The first step of evaluation is the development of a realistic test environment. We use real-world traffic datasets and publicly available infrastructure information to generate a realistic WAND topology.

#### Workloads

We integrate several real-world workloads to generate the topology and the traffic in WAND.

**(a) Cellular dataset:** The cellular dataset is obtained from a large cellular provider in China and consists of utilization information at 50 virtualized DCs over a period of 5 days.

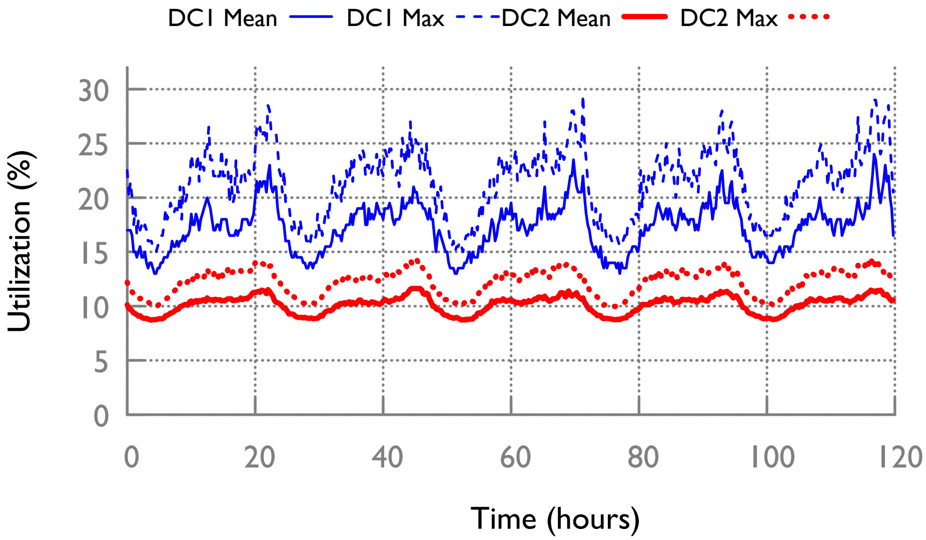


Figure 4.2: Sample traffic: Cellular dataset

At each DC, it provides information on (a) the mean and the maximum CPU utilization per server, and (b) total data sent and received from the DC. The utilization values are monitored every 10s by the cellular provider. The dataset contains the mean and the maximum at 15 min intervals over this data. The utilization at two locations are plotted in Figure 4.2. In Figure 4.4, we plot the CDF of mean and maximum CPU utilization across all 50 locations on all days. We observe that the utilization is approximately between 10% and 30%. The distribution of normalized DC sizes (# servers) is plotted in Figure 4.6). The number of servers range from $< 10$ to hundreds. The largest DC has more than 1000 servers.

**(b) DNS dataset:** This dataset contains DNS queries received by a single DNS provider at locations across the globe over a period of 24 hours (508GB of compressed data). Entries include the timestamp (at microsecond granularity) and the origin zipcode of each DNS request. From this global dataset, we filter requests originating in the continental US, which include 17,440 origin zipcodes with more than 20 billion queries in the 24-hour period. The number of queries are aggregated at minute-scale and load at two zipcodes are plotted in Figure 4.3. The CDF distributions of minimum and mean with respect to the maximum load are shown in Figure 4.5. We observe that the DNS queries have higher variability compared with the cellular dataset. The maximum load at a DC can be more than 10× the mean in certain cases.

**(c) Video camera dataset:** Video processing at the edge is a most prominent application that will benefit from WAND deployments. The number of security cameras in the US in 2016

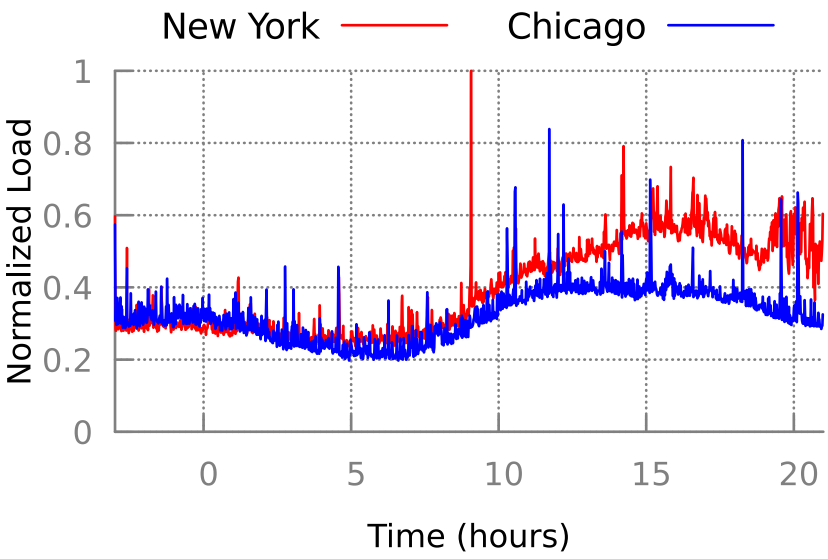


Figure 4.3: Sample traffic: DNS dataset

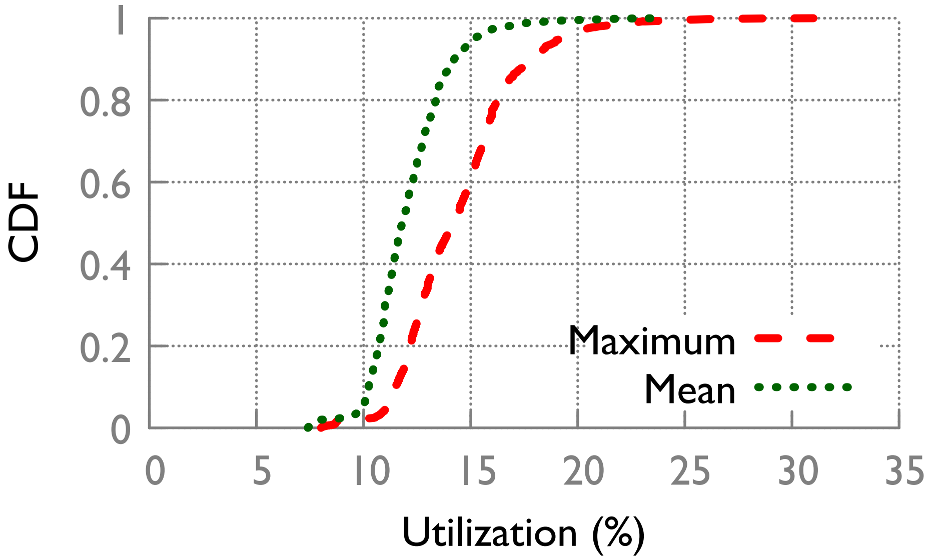


Figure 4.4: Traffic Characteristics: Cellular dataset

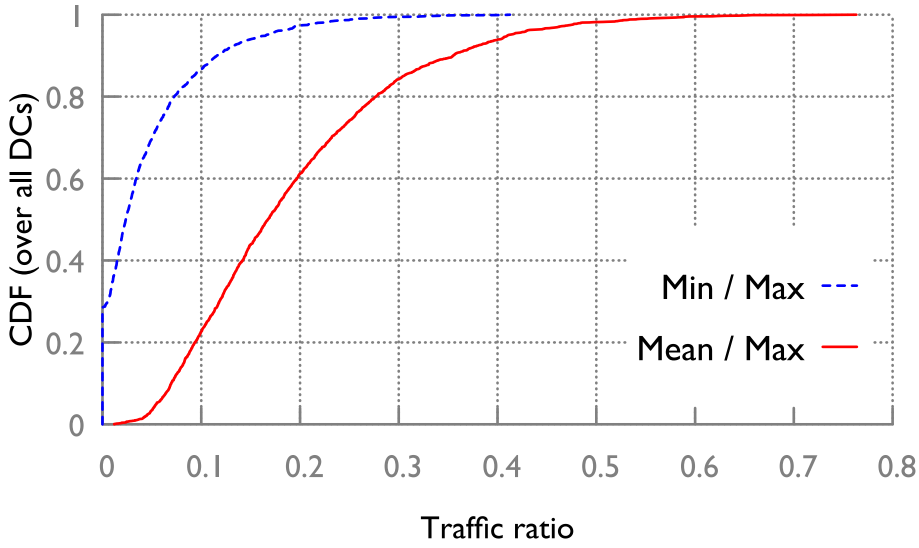


Figure 4.5: Traffic Characteristics: DNS dataset

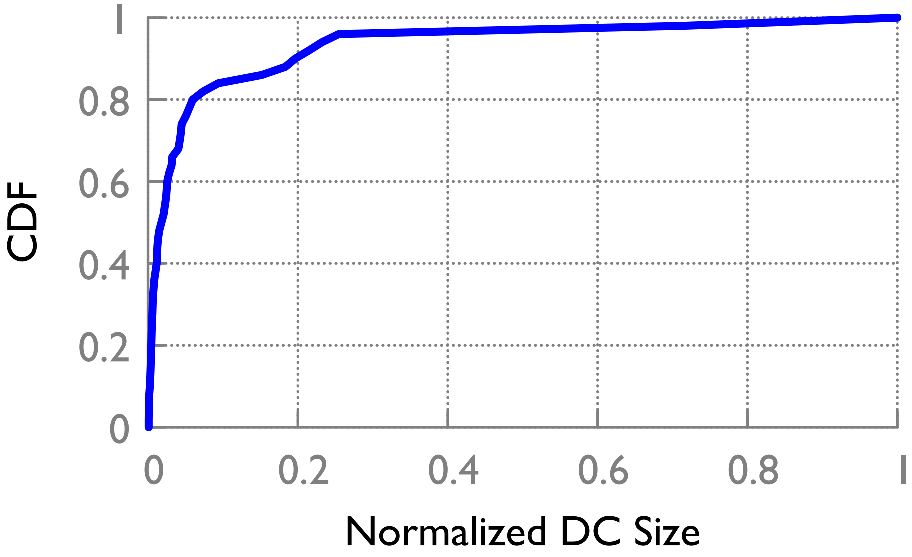


Figure 4.6: DC Size Distribution: Cellular dataset

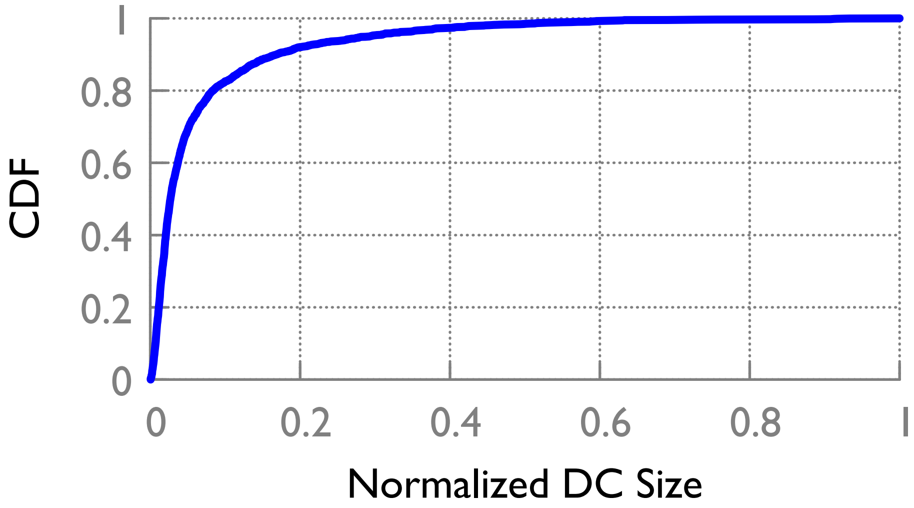


Figure 4.7: DC Size Distribution: Simulation Environment

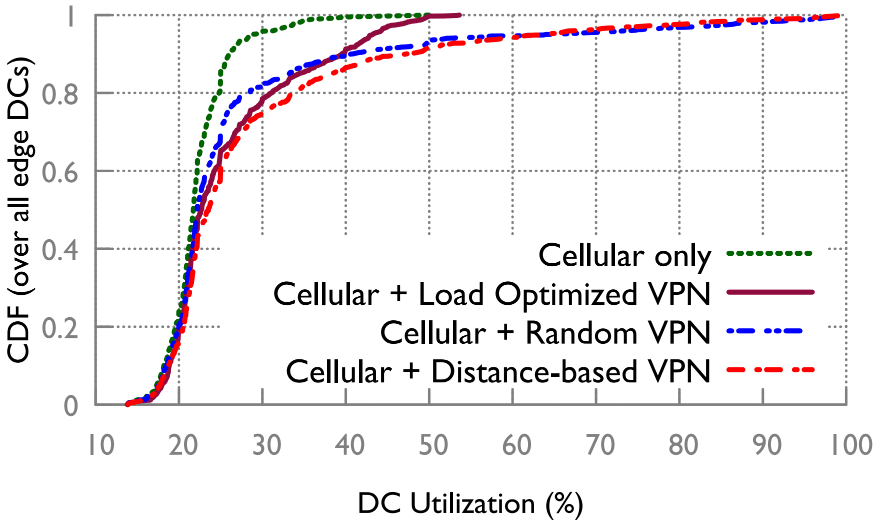


Figure 4.8: Comparison of Patronus optimization, random placement, and nearest-DC placement on VPN (Highest priority).

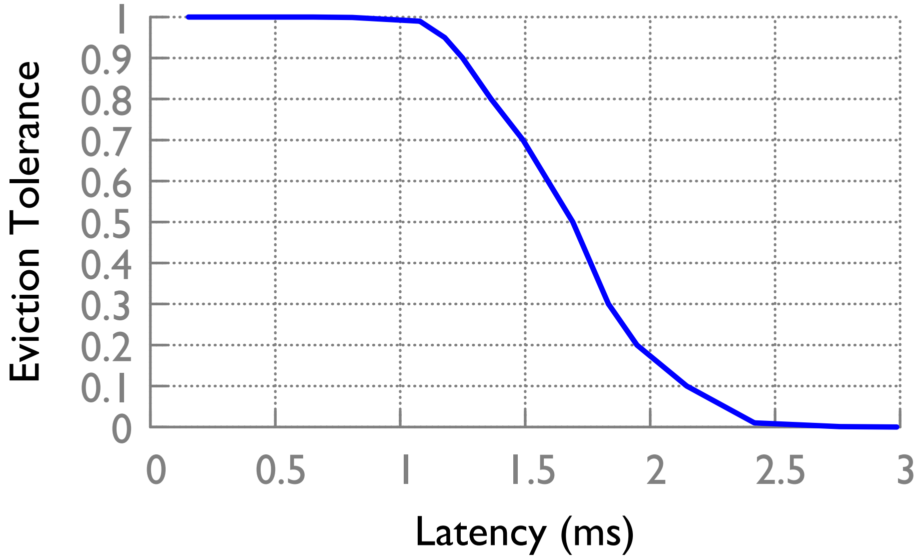


Figure 4.9: Eviction Tolerance vs. Latency (latency based on geodesic distance)

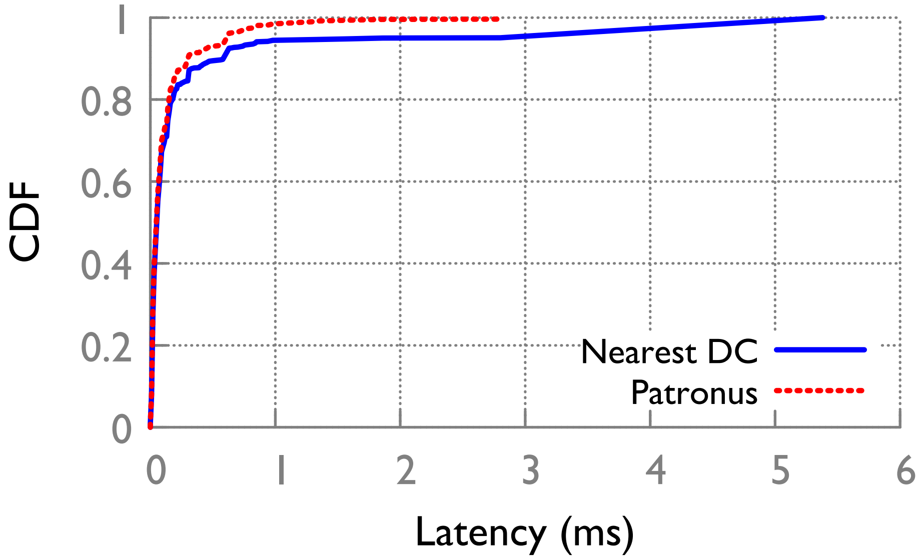


Figure 4.10: Geodesic distance-based latency for distributed video processing based on DC placement ($3^{rd}$ priority class)

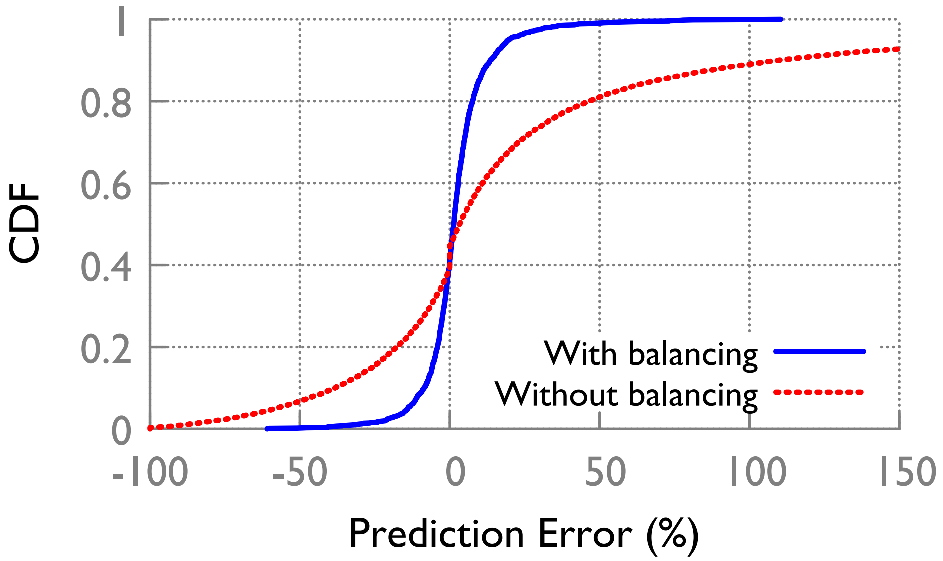


Figure 4.11: DNS dataset Prediction Error. With regional balancing of load, impact of error mitigated.

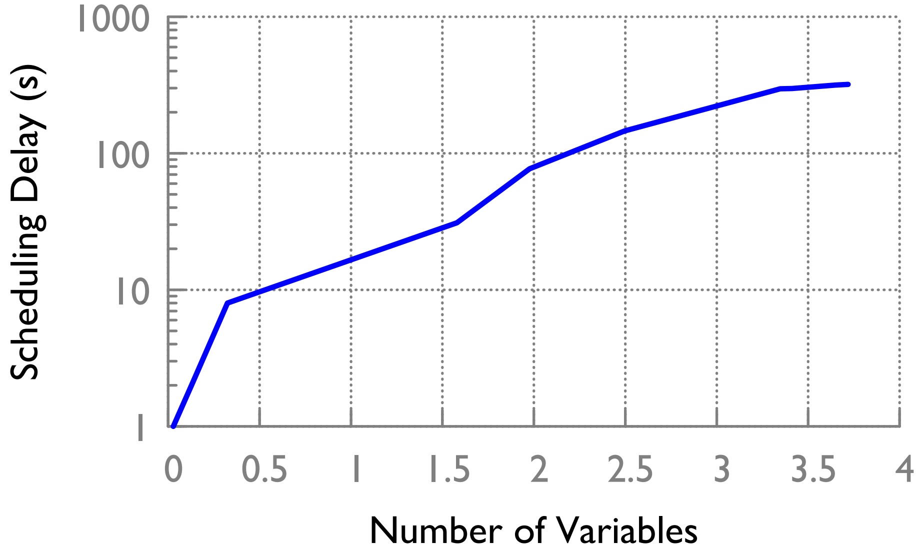


Figure 4.12: Scheduling delay as a function of number of variables (in millions).

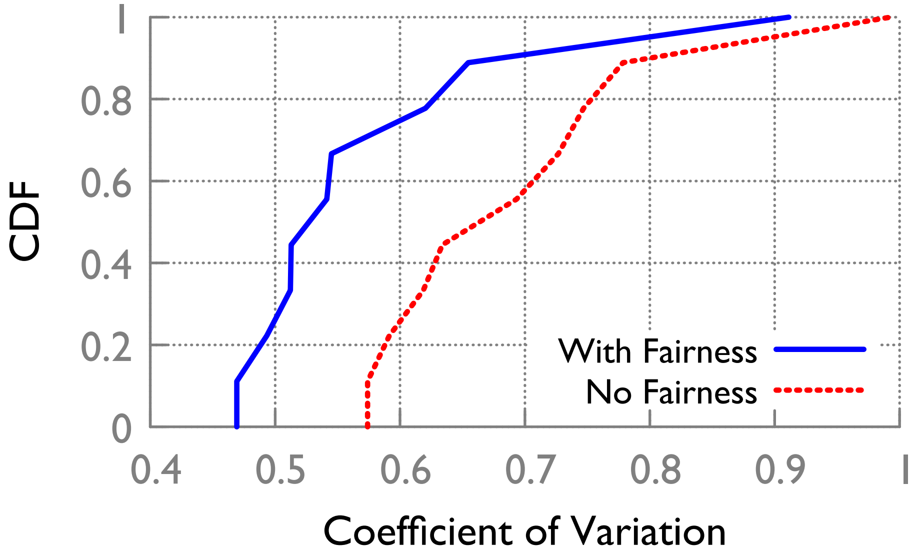


Figure 4.13: Coefficient of Variation of penalty (resources scheduled in cloud due to lack of edge resource)

is estimated to be 62 million [101]. We use the locations of red-light and speeding cameras in the US available in a public GPS forum. This dataset includes 4932 cameras across the continental US. While these cameras take snapshot pictures during traffic violations, we use this as a realistic proxy for video camera locations.

**(d) Big data analytics:** We use publicly available TPC-DS microbenchmarks [102] with synthetic location constraints to simulate batch analytics workloads in WAND. We use traces of 16 TPC-DS queries originally run on 20 machines.

### Generating the Test Environment

We run the Patronus controller on a Dell PowerEdge R440 machine with 32 cores and 128GB RAM.

**(a) Topology:** We generate a realistic WAND topology based on publicly available information on cellular infrastructure. The locations of MDCs are determined using Central Office (CO) information of a large cellular provider in the US [103]. We obtain 1864 Central Office locations associated with the 17440 zip codes in the DNS dataset.

Each MDC is assigned a capacity which is proportional to the population associated with that zip code (2016 population [104]). The distribution of data center sizes is shown in Figure 4.7. We note the MDC sizes based on population in the US and the real size distribution in China (Figure 4.6) follow similar distributions. The number of servers in the

US simulation environment ranges from 2 to 2200. In line with regional offices of cellular providers, we also add 15 regional DCs (publicly available locations of colocation centers of the same cellular provider in the US) with capacity proportional to the sum of MDCs that are closest to it.

The core topology of the same cellular provider is obtained from the Intertubes dataset [105] and has 116 nodes and 151 links. Each DC is connected to its closest core node. Conversations with cellular operator reveal that currently cellular MDCs follow a hub-and-spoke-like model with MDCs connecting to the core. With 5G, MDCs are expected to have more interconnections among them. Currently, we do not consider such interconnections in our test environment. The bandwidth of the outgoing link of an MDC is set proportional to its capacity.

**(b) Workload:** The cellular data for the test environment is obtained by applying the percentage of utilization from a randomly chosen DC in the Chinese dataset to the DC. These mapping are done a priori to obtain the timeseries for a day. VPN/other high-priority applications of cellular provider is generated synthetically. We generate 100 such applications, each with a random number of sites between 3 and 15 with a randomly chosen load at each location. These are streaming applications where all sites are concurrently active.

The scaling factors for datasets are determined proportional to their loads in such a way that the load (as number of containers) is neither too low nor too high at most DCs. For the DNS dataset, the load in containers is proportional to the number of requests. For the camera dataset, the number of containers is proportional to the cameras assigned to a DC.

### 4.5.4 Results

We evaluate the ability of Patronus controller to utilize the resources efficiently and meet the application requirements.

#### Efficiency of Hierarchical optimization

VPN and other applications which belong to the highest priority class are slightly more flexible than the cellular traffic. We assume that the cellular traffic need to be processed at the nearest MDC (when resources are available) while the VPN allocation may be at any MDC within 1ms from the origin of the traffic. Hence, to determine long-term placement for non-cellular high priority applications, we run a two-level hierarchical optimization at the peak period of cellular load.

The optimization has 2 objectives. The highest priority objective is shortest path allocation of cellular traffic. In Figure 4.8, we observe that shortest path allocation on cellular traffic load results in a maximum edge DC utilization of 50%. For the VPN traffic, we evaluate several techniques. Minimizing the maximum load of DCs with latency-based placement constraints in Patronus can accommodate all the non-cellular applications while maintaining the maximum utilization at 53.6%. On the other hand, placing the VPN load at the closest MDC to the origin or at a randomly chosen MDC within the prescribed radius (1ms) leads to high utilization in some DCs. Note that jointly optimizing the cellular and VPN loads may result in some cellular traffic being redirected to non-shortest DCs. Hence, multi-level optimization may be essential even within the same priority class.

**Eviction Tolerance/Latency Trade-off**

The trade-off between eviction tolerance (4.4.2) and latency at a single MDC in third priority class is given in Figure 4.9 (first priority is cellular and VPN applications, second priority is DNS workload). We consider the video analytics application load in third priority class at a single MDC. Eviction tolerance is computed based on the mean and standard deviation of total traffic in the higher priority classes for an interval of 15min. The figure shows mean latency over a period of 19 hours at each data point. When the eviction tolerance is the highest at 1.0, the latency is 0.15ms. As the eviction tolerance decreases, the latency of the application increases. At eviction tolerance value of 0.7, the latency increases 10× to 1.5ms. When the application is very conservative ($\theta < 0.1$), the latency increases to approximately 3ms.

**Latency of applications**

We measure the latency perceived by the video analytics dataset in priority class 3 (Figure 4.10). We compare Patronus optimization with nearest-MDC placement. In both scenarios, when an MDC is not available, the task is placed on the nearest regional DC. We observe that the tail latency is more than 2× lower in Patronus. Note that the latency is computed based on geodesic distance here. With additional infrastructure overhead, the differences will be more pronounced since Patronus places tasks at the edge at a higher rate.

**Prediction Efficiency**

We evaluate the predictability of the two real-world datasets. In the sparse cellular dataset, we use statistical measures. We consider two scenarios: (a) past hour on the same day, or (b) Time of Day (ToD) in past $n$ days. When the prediction is defined as the maximum observed load, ToD of 4 days is a better predictor. Over a day, prediction based on recent history can lead to under-prediction up to 6.5% and over-allocation up to 8% while ToD prediction limits the errors to 3.1% and 6% respectively. Hence, we use ToD demand of 4 days with a 5% over-allocation to meet the demand.

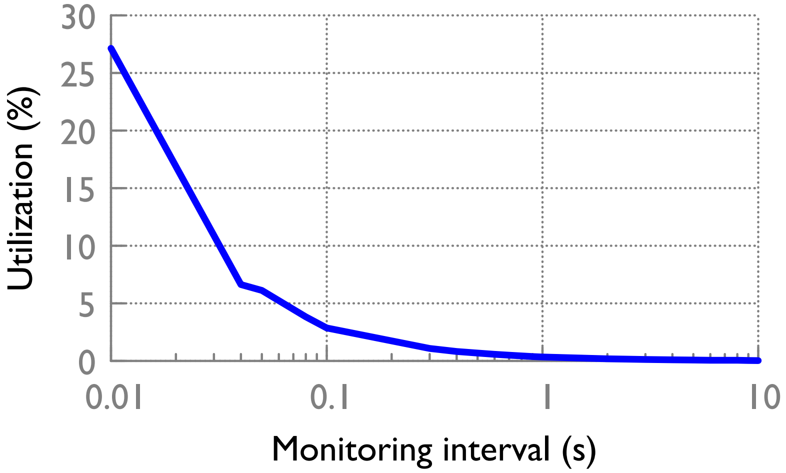


Figure 4.14: Performance monitoring overhead at various monitoring intervals: CPU utilization

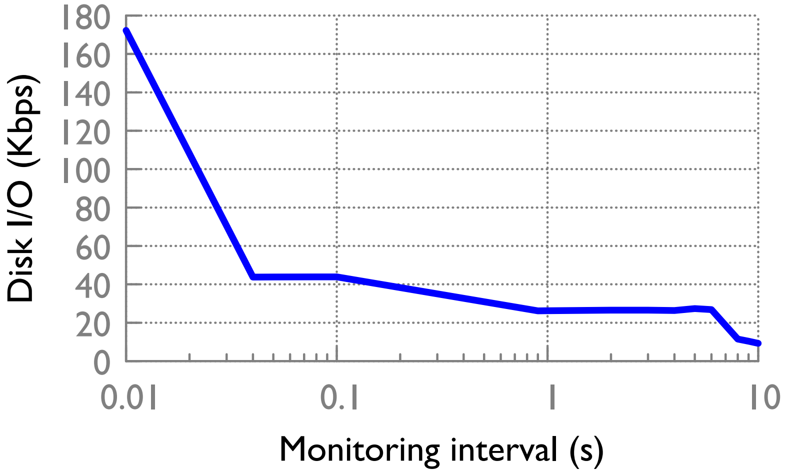


Figure 4.15: Performance monitoring overhead at various monitoring intervals: Disk I/O

In the DNS dataset, the error in prediction using NN is given in Figure 4.11. In the dataset with samples every minute, data from past 10 min is used to predict the expected load as number of DNS requests in the current instance. The error is high due to high variability in the workload (shown in Figure 4.5). However, if the application can effectively load-balance across MDCs in an area, the error reduces considerably. This shows that applications with high load variability can benefit significantly from load-balancing at the edge.

**Scalability**

We test the scalability of both hierarchical and long-term schedulers in Patronus resource allocation module. Since both the schedulers rely on hierarchical optimization, we plot the scalability in terms of number of variables (Figure 4.12). The scheduling latency was measured with 4 optimization objective priority levels in the optimizer.

The number of variables depends on a variety of factors: number of MDCs and links, number of applications, number of tasks in batch jobs, number of user-groups in streaming jobs, number of timeslots, etc. Even with millions of variables, the scheduling latency is in seconds. Note that the number of variables in hierarchical optimization is 2-3 orders of magnitude smaller in instantaneous scheduler compared to the long-term scheduler. For example, if we consider 1000 DCs and 100 applications with a components in every DC, we have approximately 0.1 million variables. In hierarchical optimization, not all of them may be active at the same time, reducing the complexity even further. In this range, the optimization takes seconds. The corresponding long-term scheduler with a one-day plan has 0.1 million $*1440$ variables.

**Fairness**

We compare the fairness achieved with Patronus fairness scheme compared with a scheme where no fairness is enforced (arbitrary sharing of resources across applications in the same class determined by the LP solver). We assume that for batch applications, when the edge resources are unavailable, they can run in the cloud, but this is expensive. We define the penalty as the amount of resources allocated on cloud when edge resources are not available. In Figure 4.13, we observe that the coefficient of variation of penalty across applications (a heuristic for unfairness) is reduced with fair-sharing in Patronus.

**Performance monitoring**

We measure the overhead associated with monitoring using a lightweight Linux module which measures applications' usage of CPU, memory, network, and disk. The CPU and disk utilization of the monitoring module at various monitoring intervals are given in Figure 4.14 and 4.15 respectively. The overhead is negligible ($< 0.5\%$ in CPU and 9Kbps in IO) when monitoring interval is 1s.

To summarize, we make the following contributions in the inter-data center context:

- We identify control challenges in an emerging dynamic environment of interconnected Micro Data Centers which we refer to as WAND (WAN As A Network of Data Centers).
- We build Patronus, an automated control system, for efficient resource management in WAND.
- We address the scalability challenge by partitioning the scheduling problem into two temporally: instantaneous scheduling handling immediate allocation and long-term scheduling for meeting critical application requirements across time.

- We show that Patronus is scalable, resource-efficient with balanced load across MDCs, and capable of meeting stringent application requirements.

# Chapter 5: INTENT-AWARE APPLICATION NETWORK INTERFACE

To achieve peak application performance, it is necessary to translate the application's high-level performance needs, or intent, to network-level requirements that are actionable for a network controller. This intent is expressed to the controller through a representation that we refer to as the Application Network Interface (ANI). The expressiveness of the ANI can affect application performance significantly.

ANIs and the flexibility they offer have evolved over time. The earliest congestion control and traffic engineering schemes focused on simple proxies for application performance at *packet* level — throughput and per-packet delay and jitter. Rate Control Protocol [12] made a step towards application-level performance goals with *flow* as the ANI and emphasis on Flow Completion Time (FCT) or the time of arrival of the last packet. Another significant leap towards an ANI that captures the requirements of distributed applications was the *coflow* [13]. Inspired by cloud applications such as MapReduce, coflow considers a set of parallel flows within an application as a single entity where the FCT of the last flow determines the performance. This enables scheduling schemes to borrow bandwidth from lighter flows in the coflow to speed up the heavier flows, thereby improving application deadlines.

We observe that even the coflow abstraction is insufficient to support requirements of today's sophisticated applications. Applications such as distributed deep learning and interactive analytics have a complex interplay of communication and computation at the participating nodes. In this scenario, not all flows within a coflow are equivalent from the perspective of the application. Depending on the nature of computation, the application may benefit by finishing some flows sooner than others within a coflow. For example, multiple parameters are exchanged between the parameter servers and a worker in distributed deep learning. These parameters are consumed at different times based on the underlying computational model in frameworks such as TensorFlow. Hence, the iteration time can be improved significantly when the relative priorities of flows (parameter transfers) are made known to the network controller. Different applications may have other dependencies that require metrics such as relative weights or deadlines. More importantly, an application may have an explicit optimization objective different from minimizing the completion time that cannot be conveyed through current ANIs.

In this paper, we argue that it is an opportune moment for narrowing the gap between application intent and its network representation through a more expressive ANI for cloud applications. The stringent performance *needs* of cloud applications coupled with the *opportunity* to extract fine-grained application characteristics using sophisticated learning tech-

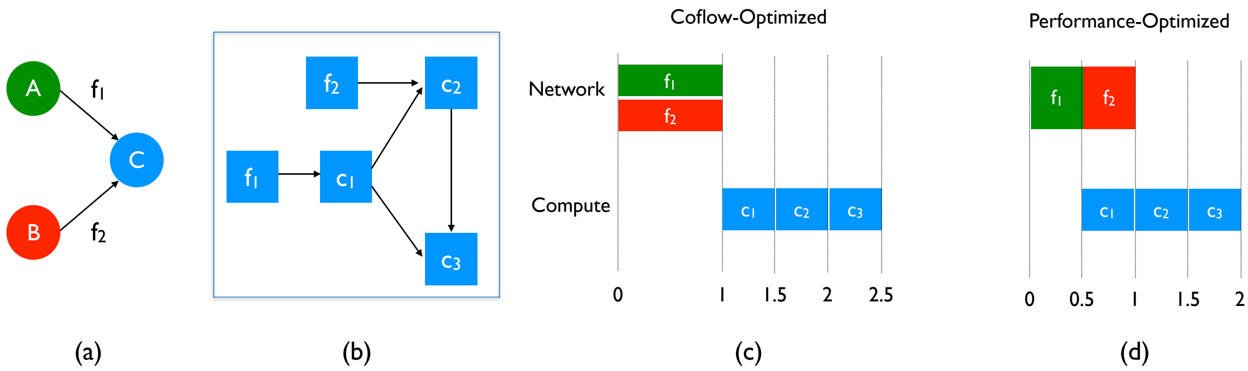


Figure 5.1: Importance of understanding application intent: (a) Coflow with two component flows. $f_1$ and $f_2$ (size 500Mb each) share a 1Gbps bottleneck link. (b) Computation model at $C$ has 3 operations, $c_1$, $c_2$, and $c_3$ with dependencies between flows and computations as shown. Each computation operation takes 0.5s to execute. Completion times with (c) coflow completion time-optimized transfers and (d) intent-based optimization for transfers.

niques inspire us to rethink cloud ANI design. First, we analyze several cloud applications to understand communication patterns that are not captured by current ANIs. Second, we quantify the performance benefits achievable with a more expressive ANI in a popular application, distributed deep learning. We show that iteration time in deep learning training can be improved by up to 25% using additional information made available through CadentFlow. In a shared network environment, the improvements increase further up to 46%. Finally, we put forward an application intent-aware ANI and discuss its implication for cloud systems design.

We introduce CadentFlow, an ANI whose semantics include component flows of an application with associated per-flow metrics, and an optimization objective for capturing the application intent. A more expressive abstraction opens several research directions. (i) *Extraction of CadentFlow attributes* such as per-flow metrics and the optimization objective requires analysis of applications' models or learning-based inference. (ii) *Redesign of network controllers* with novel scheduling algorithms that can leverage the richer semantics of CadentFlow to provide performance guarantees compliant with the application intent. (iii) *In-network implementation* of intent-aware scheduling schemes paves the way for exploring the flexibility of programmable switches to implement application objectives.

## 5.1 UNDERSTANDING THE ENVIRONMENT

In this section, we demonstrate the need for rethinking ANI in the cloud environment where multiple applications share the network. First, we illustrate shortcomings of state-of-the-art ANIs. Second, we examine distributed applications that can benefit from an improved ANI.

### 5.1.1 The application perspective

Large-scale computations in data center environments involve multiple concurrent flows with such data transfers accounting for more than 50% of the job completion time [106]. The coflow abstraction [13] was introduced to consider the correlated flows as a single entity, thereby enabling collective optimization on the set of interdependent flows. However, as cloud applications evolve this ANI may be insufficient.

For example, not all transfers may be of equal importance to the computation in the succeeding stage of an application. The data in a single stage maybe consumed in a particular order, may require weighting according to application needs, or some chunks may have tighter deadlines. In this case, the application will benefit when flows are prioritized/weighted based on its intent. For example, consider the simple coflow in Figure 5.1a with two components flows of equal size: $f_1$ and $f_2$. The computation at node $C$ involves 3 tasks with dependencies as shown in Figure 5.1b. Since the first stage of computation depends on $f_1$, the application performance can be improved by prioritizing packets belonging to $f_1$. The application completion time with coflow optimization, in this case, is 2.5s while the optimal time is 2s.

While this simple example can be solved by splitting the single coflow into two ($f_1$ only, $f_2$ only) and adding dependency between them (similar to multi-stage DAG scheduling in Aalo [107]), this approach cannot be generalized for several reasons. First, coflow dependencies cannot be determined apriori in certain systems (e.g., graph processing systems operating on time-evolving graphs). Second, some applications may prefer weighting across flows instead of strict ordering enforced by a DAG of coflows (e.g., stream processing systems). Third, coflow completion time (CCT) is a proxy for application performance, job completion time. As we show in § 5.3.2, a lower CCT may not always correlate with better job completion time. Finally, in multi-application environments, significant performance benefits can be achieved with deadlines (§ 5.3.2), a benefit not achievable with a DAG of coflows. Moreover, an application may have other explicit objectives and finer-grained preferences that are not captured by flow/coflow completion time.

While prior work has considered inter-flow relationships [13, 108, 107, 106, 109, 110] and deadlines [111, 112] in the cloud environment, the set of objectives handled by the network controller has been limited, often to a single objective across all applications. Inter-coflow scheduling schemes [108, 107, 106], even those which are information-agnostic [113, 110], primarily focus on a single objective, minimization of mean CCT. Prior solutions lack an expressive API that supports complex objectives of advanced data flow systems.

It is also worth noting that application level order enforcement alone at the edge is not sufficient since flows originating at multiple nodes often need to be coordinated.

### 5.1.2 Which applications will benefit?

We analyze different families of advanced data flow systems to understand their requirements.

**Distributed Deep Learning:** This workload has iterative computation involving multiple workers and Parameter Servers (PSs). The parameters, updated at the end of each iteration, are acted upon by the worker nodes at different time instances determined by the underlying computational model in frameworks such as TensorFlow [114] and PyTorch [115]. In this case, the iteration time can be improved by prioritizing parameter transfers in the order in which they are consumed.

**Partition-Aggregation:** In online services including Web page delivery and search query responses, the requests from users are partitioned across multiple workers in the back-end. The results are aggregated by the front-end server/proxy and sent back to the user. Many services begin sending a response before the complete response is available at the server. In this scenario, the application may benefit by prioritizing those flows which are critical to the response. For example, a web proxy waiting for components in a web page can choose to delay fetching a large image. While application-level solutions exist [116, 117], a controller with visibility into multiple such applications can further improve performance across them (similar to § 5.3.2).

**Graph Processing Systems:** In graph processing systems [118, 119, 120], the graph is partitioned across multiple nodes which process vertices in a sequence and exchange the results of computation after each iteration. In this scenario, when a node receives information from multiple neighbors, those flows whose information will be processed first may be accelerated to improve the iteration time. In systems which handle time-evolving graphs [121, 122], these priorities may change over time.

**Interactive Analytics:** Big data systems analyzing real-time low-latency queries (e.g., Naiad [123]) incorporate explicit application deadlines on each stage of the data flow. This provides opportunity for accelerating/slowing down the network transfers based on deadlines. In stream processing systems [124, 125, 126] with load balancing across parallel components, the end-to-end performance may be improved by weighting the flows according to application preferences.

**Teleimmersion:** In teleimmersion systems with multiple cameras, there are multiple flows/stream bundles with co-dependencies. In this environment, prioritization of flows directly affects the immersion experience of users[127, 128, 129, 130, 131].

**Batch Processing Systems:** MapReduce [2] framework and others which use it as a building block [132] have hard barriers between stages. In this case, the coflow abstraction would suffice to capture application requirements. In batch processing systems without hard barriers (e.g., MapReduce Online [133]), the application performance may be improved by prioritizing flows which are more important to the next stage of computation.

Thus, in a variety of scenarios, intent-awareness helps in prioritizing/accelerating those flows which are critical to the application. The coflow abstraction is sufficient when the application has *hard barriers* after each stage. However, several common distributed applications have more complex workflows, inspiring us to rethink the network interface for cloud applications.

## 5.2 INTENT-AWARE ANI

In this section, we identify the critical components necessary for an intent-driven ANI and put forward CadentFlow. Analyzing a wide range of cloud applications, we identify two essential features missing in state-of-art ANIs.

**Missing pieces in current ANIs:** The first missing piece in today's ANIs is a means for explicitly conveying application's intent. For instance, the only application-level metric currently supported by coflows is the minimization of coflow completion time. However, applications may have other performance objectives such as maximizing an application-specific utility in terms of per-flow completion time and bandwidth. Hence, we argue that the optimization objective should be an explicit part of the ANI.

The second deficiency in state-of-the-art ANIs is the inability to represent complex dependencies across flows (beyond membership in a set defined by coflows). Dependencies across coflows have been considered with inter-coflow DAGS [107, 134]. However, in practice, dependencies may take several forms. For example, an application may require weighted splitting of bandwidth among its component flows or may have deadlines per flow.

**Defining CadentFlow:** Having identified the key missing pieces in an expressive ANI, we put forward CadentFlow, an ANI with an application-level objective and per-flow metrics, for effectively capturing intent.

A *CadentFlow* is a set of correlated flows between a collection of machines with an application-level optimization objective denoted by $\Gamma$ and a set of tagged flows. Each component flow, $f_i$, has an associated list of tags, where each tag is a tuple with metric type and metric value. $T_i = (t_{i1}, m_{i1}), .., (t_{ik}, m_{ik})$. A CadentFlow, $CF$, can be represented as: $CF = \{\{(f_i, T_i), (f_2, T_2), ....., (f_n, T_n)\}, \Gamma\}$. We propose weights, deadlines, and priorities as preliminary candidate metrics for denoting the inter-dependencies between flows. As applications and their requirements evolve, more metrics may be added to this set.

**CadentFlow Representation of Applications:** We present CadentFlow representations of applications discussed in § 5.1.2. (i) *Distributed deep learning:* There are two possible representations for DNN training. (a) Frameworks such as TensorFlow provide application-level DAG with inter-dependencies between communication and computation. This information can be used to determine priorities of flows, and the objective will be minimizing completion time of last flow, subject to scheduling based on priorities. (b) Since, DNN training is an iterative process, we can estimate the time required by computation operations in a given system. Combining this information with the DAG, deadlines can be estimated for individual flows. This provides additional scheduling flexibility. The objective is minimizing maximum delay subject to deadlines for all flows. This representation improves flexibility by allowing delayed scheduling of flows with flexible deadlines (as shown in § 5.4(b)).

(ii) *Partition-Aggregation:* In Web page delivery, priorities can be set based on rendering preferences. The objective is minimizing completion time subject to prioritized transfers. (iii) *Graph processing systems:* Weights used as metrics for load-balancing, objective is minimize completion time subject to weighted transfers. (iv) *Naiad (interactive analytics)*: Deadlines used as metrics. Objective is minimization of sum of delays with respect to deadline. (v) *Batch processing with soft barriers*: Adaptive priorities based on application progress used as metrics. Objective is minimization of completion time.

## 5.3 EXPERIMENTS

We quantify benefits achievable with intent-awareness in CadentFlow using distributed deep learning as a representative application.

### 5.3.1 Methodology

**Workload:** We test deep learning workloads using TensorFlow under two scenarios: DNN training and inference. In the training workload, during each iteration workers (each with an identical copy of the model) send updates on parameters to the Parameter Servers (PS). PS aggregates the changes and returns the updated parameter to all workers. In the inference workload, inference agents read the parameters from the servers and run the inference.

This captures the online inference scenario where agents (separate from the workers and Parameter Servers) read the latest version of parameters during reinforcement learning and serve inference queries.

The TensorFlow model is a DAG composed of dependencies between two types of operations (ops): computation and communication ops (parameter transfers). Note that in training and inference, not all parameters are consumed at the same time (e.g., parameters of layer 1 are used before layer 2). We test 11 popular Neural network models [1] (including AlexNet [135], Inception-v1 [136], Inception-v3 [137], ResNet [138], and VGG [139]) on TensorFlow 1.8 with Parameter Servers using the standard batch size for each model.

**Estimating metrics:** We evaluate the workloads under two scenarios: CadentFlow with (a) priorities as flow metrics and (b) deadlines as flow metrics. The TensorFlow model with computation-network dependencies is readily available through an API. We analyze this DAG to estimate priorities in (a). While (a) is relatively easy, for (b), we add tracing to the TensorFlow system. We collect the runtime information on Standard NC6 virtual machines (6 cores, 56 GB RAM, 1 X Nvidia K80 GPU with 12GB RAM) in Azure. We estimate the time taken by computation ops by running the same model 10 times on a single machine and taking the mean.

Deadlines are derived using the traces. Recall that training has two phases: (a) forward pass where latest parameters are read from PS and loss function is computed, and (b) backpropagation phase where parameters are updated and sent to PS. Thus, we have two CadentFlows in one iteration. In the forward pass, all flows are assigned the same estimated start time at 0 (all parameters are ready for transfer at PS). The deadline of a parameter is computed as the total computation time before the read operation of that parameter. Thus, the deadline for a parameter flow depends on its position in the DAG, i.e., the total computation time of its dependencies. In the second CadentFlow, the estimated start time is computed as the total computation time before the parameter is ready to be sent to the PS. All flows share the same deadline determined by the end of computation. The inference workload only has forward pass and hence a single CadentFlow with deadline estimation similar to training forward pass.

Note that the model at all workers is identical. The deadline estimation does not take into account available bandwidth. The deadlines represent the preferences of the application and the flexibility available between flows from the computation perspective.

**Control Schemes:** In our experiments, we test 3 schemes: (i) TCP simulated with max-min fair sharing across flows sharing a link, (ii) Coflow scheduling using Minimum Allocation for

[1] https://github.com/tensorflow/models/tree/master/research/slim

Desired Duration (MADD) used in Varys [108] where lighter flows are allocated a lower bandwidth in such a manner that all flows in a coflow finish at the same time, (iii) intent-aware CadentFlow scheduling where flows are transferred based on the optimal order/deadline determined from the TensorFlow model. For a fair comparison, we assume that the baseline TCP connection transfers the parameters in the best possible order from a given node. However, TCP cannot enforce inter-flow priorities across multiple TCP connections from different Parameter Servers to the worker.

We simulate various schemes with multiple configurations (1 to 16 PS/workers) where each host has 1/10 Gbps NIC. For Coflow and CadentFlow scheduling, we assume a centralized controller with global view which can make globally optimal decisions and enforce them at the edge. We evaluate performance benefits achievable with a single active application as well as multiple applications in the network.

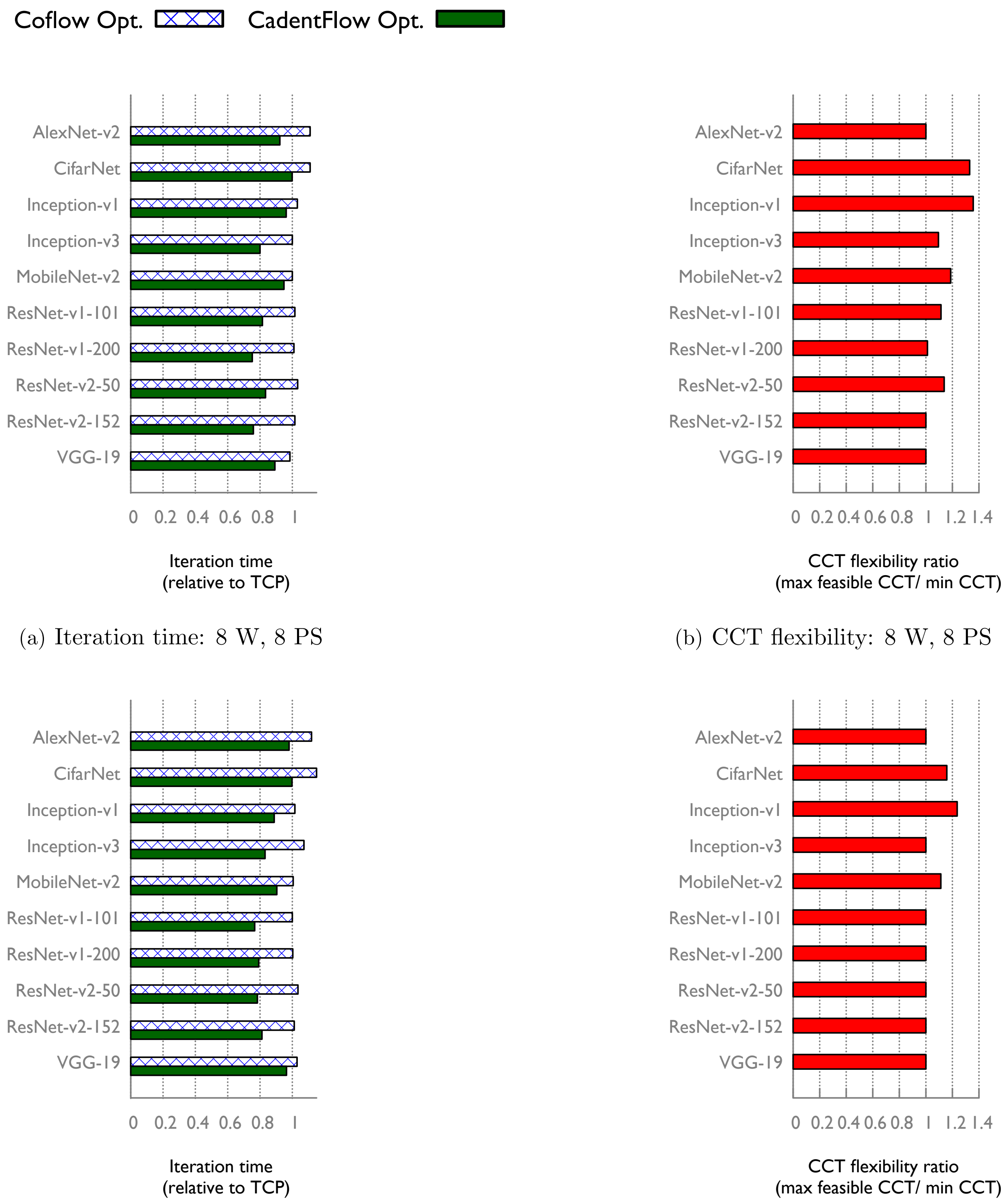
**Application Simulation:** We measure two metrics on the distributed deep learning application: the iteration time and the CCT flexibility ratio. The ***iteration time*** is the time taken by one complete iteration composed of computation time (empirically estimated) and communication time (evaluated using the three control schemes). In the training workload, this includes the computation in forward pass, backpropagation phase, and two CadentFlows (from the PS to worker before forward pass and from workers to PS after the backpropagation). In the inference workload this is composed of one CadentFlow for fetching parameters from the servers and the inference computation. We assume that all the workers have identical computation time, i. e., no stragglers.

We introduce the metric, ***CCT flexibility ratio***, to measure the available flexibility in flow scheduling when *deadlines* are used as the CadentFlow metric. The application intent-based deadlines maybe be more flexible than the minimum achievable Coflow Completion Time. This provides the network controller with the opportunity to delay some flows without affecting the application performance. We measure this flexibility in scheduling using CCT flexibility ratio. This metric is defined as the ratio of maximum deadline acceptable to the application (the latest deadline) and the minimum CCT achievable with the available network.

### 5.3.2 Results

We present results on training and inference workloads across the three control schemes in an environment with 10Gbps network. The conclusions were also verified with 1Gbps.

**Single Application** To understand the differences between control schemes, we first con-



(a) Iteration time: 8 W, 8 PS

(b) CCT flexibility: 8 W, 8 PS

(c) Iteration time: 16 W, 16 PS

(d) CCT flexibility: 16 W, 16 PS

Figure 5.2: (a,c) Coflow and CadentFlow optimizations plotted relative to TCP. Lower iteration time is better. (b,d) CCT flexibility shows the window of flexible time available for scheduling with respect to minimum Coflow Completion Time for deadline-based CadentFlow.

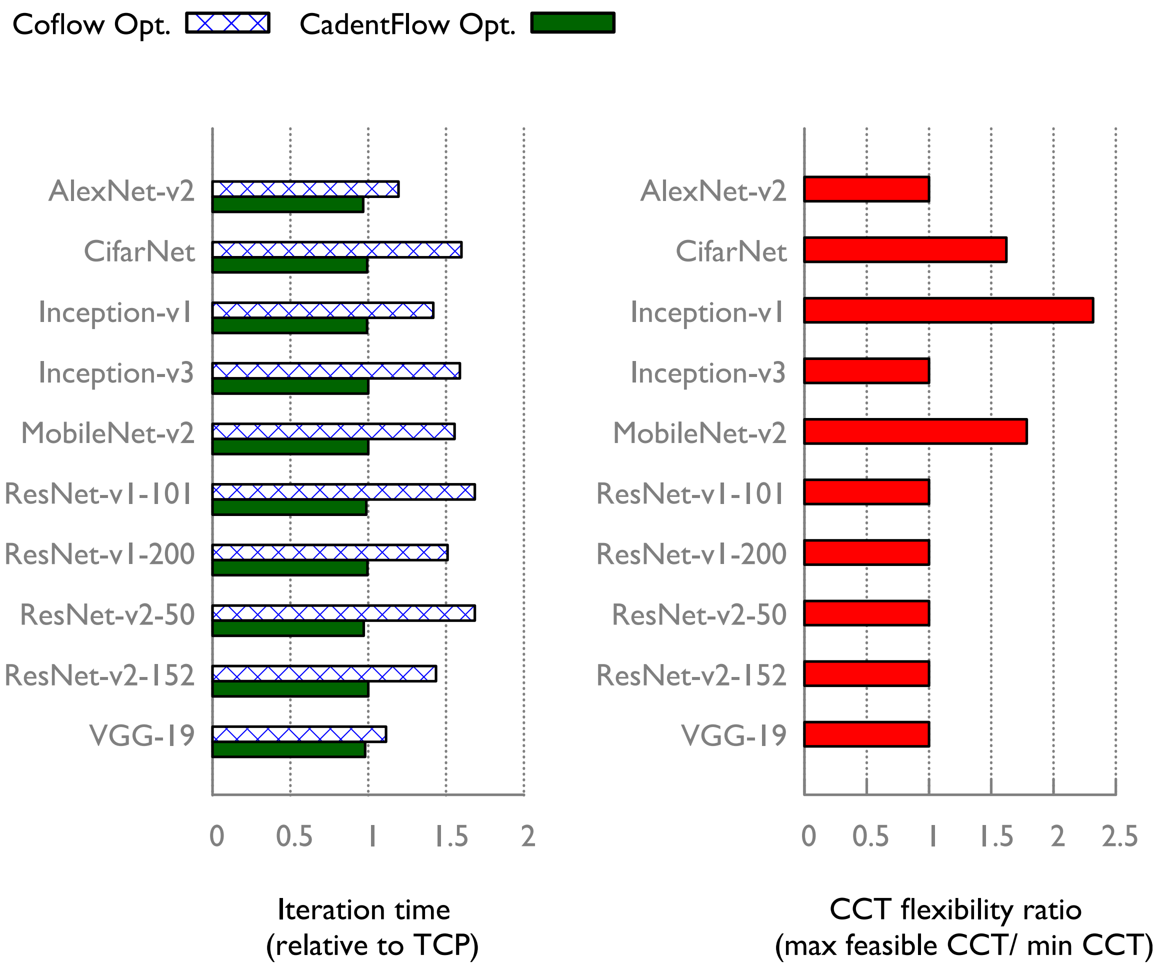
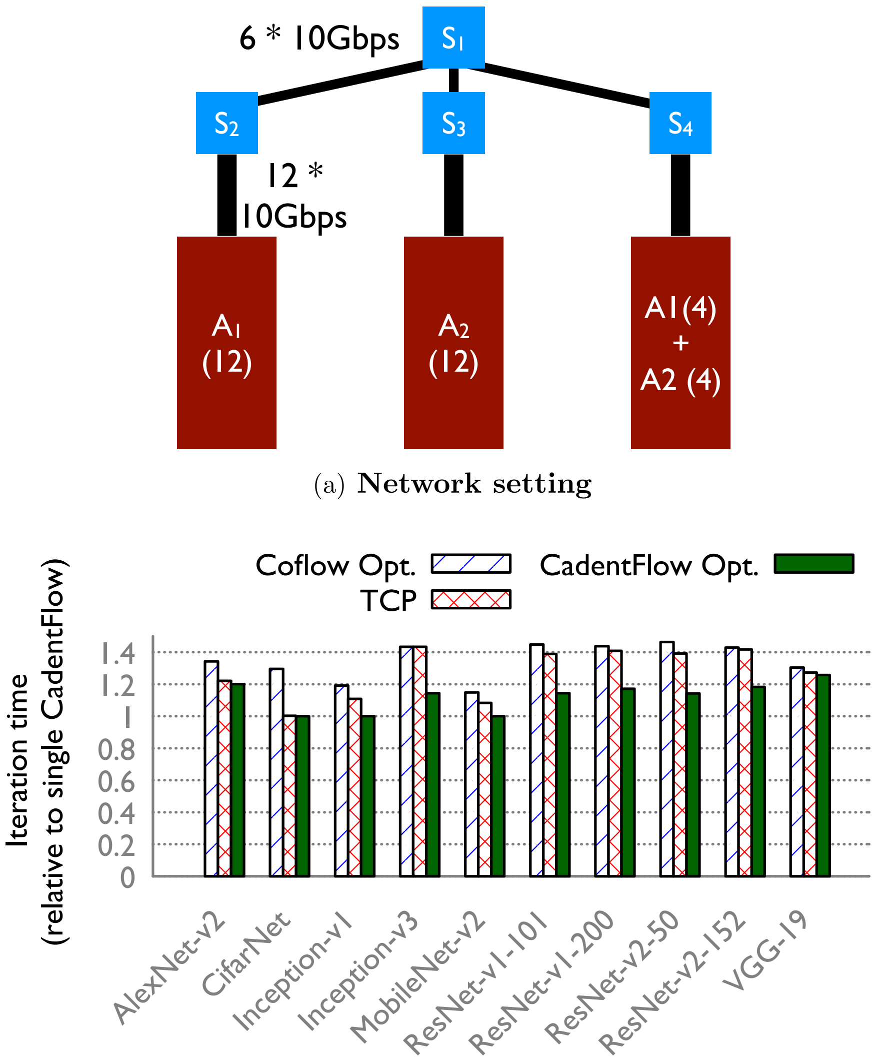


Figure 5.3: Inference workload (16 workers, 16 Parameter Servers)

duct experiments with a single active application in the network. The iteration time of coflow optimized scheme based on MADD and the CadentFlow optimized scheme are compared with the TCP baseline in Figures 5.2 (a,c) and 5.3 (a). A lower iteration time implies better performance. The iteration time results are the same with both priority-based and deadline-based CadentFlow optimization, with a maximum reduction in iteration time of 25% (ResNet-v1-200 with 8 W, 8 PS). We observe that Coflow scheduling can result in worse performance than TCP in some cases. For example, the iteration time on CifarNet is 11% higher with coflow optimization compared to TCP in a system with 8 workers and 8 PS. This is due to the coflow scheduling mechanism that delays smaller transfers to accelerate large transfers. This results in larger delays for smaller parameters that have higher priorities. Note that the CCT is same for both Coflow and CadentFlow optimizations in this scenario. However, application iteration time is lower for CadentFlow optimization. Also,



(a) **Network setting**

(b) **Performance improvement**

Figure 5.4: (a) 12 servers of rack 1 connected to switch $S_2$ runs application instance $A_1$. 12 servers of rack 2 connected to switch $S_3$ runs instance $A_2$. Out of 12 servers in rack 3, 4 servers belong to $A_1$ and 4 to $A_2$. (b) Performance improvement achievable with efficient overlap of multiple jobs in a shared network environment with deadline-based CadentFlow.

CadentFlow optimization performs at least as good as TCP in all tested scenarios.

With deadline-based Cadentflow, we can also obtain CCT flexibility (ability to delay flows until deadline without affecting iteration time) as seen in Figures 5.2 (b,d) and 5.3 (b). In some models, we obtain both improvement in iteration time and high CCT flexibility (e.g., Inception-v3 with 20% lower iteration time and 9.5% of added flexibility in coflow completion times). The flexibility depends on the ratio of time taken by computation and communication in various models. There is high flexibility when the models are computation-heavy, giving opportunity to delay transfers.

In the inference workload (Figure 5.3), coflow optimization leads to significant performance degradation (up to 68.5% increase in iteration time compared to TCP in ResNet-v1-101). However, CadentFlow performance is comparable to that of TCP with significant flexibility in flow scheduling across many networks.

**Multiple Applications** With deadline-based CadentFlow, we demonstrate benefits of CCT flexibility by running two applications in a shared network (Figure 5.4(b)). We test two instances of the same DNN model across three racks, where one rack is dedicated to each application (12 servers) and one rack is shared (4 nodes each in a rack per application). The fair-share for each application in the shared link $S_1$-$S_4$ is 30 Gbps while the bandwidth requirement is 40GBps. CCT flexibility allows us to schedule flows in a manner that maximizes 40GBps allocation to each application instance. In Figure 5.4(b), we compare the three control schemes in this multi-app environment with respect to the iteration time with CadentFlow scheduling when only a single app is active. We observe that CadentFlow scheduling has the best performance across all models due to the flexibility provided by deadlines. Coflow scheduling can result in up to 46.3% higher iteration time (as in ResNet-v2-50) since deadline information is not available for flexible allocation. Since deadline information is necessary for accruing these benefits, even a DAG of coflows cannot provide this performance improvement.

In summary, we make the following observations:

- Flow-level metrics (priorities/deadlines) in CadentFlow allow optimizing the application performance by up to 25%.
- Deadline-based CadentFlow can improve network-wide performance by up to 46% by providing increased flexibility.
- Coflow optimization can hurt application performance when there are complex dependencies. In such cases, falling back on TCP should be preferred.

# Chapter 6: FUTURE WORK

This thesis presented foundational results on improving throughput across different layers of data center infrastructure. This work paves way for solving several challenges in the future.

- **Randomization of task placement:** Our results on understanding physical layer throughput shows that randomization of traffic at the rack level using realistic traffic matrices can improve throughput performance. Can we leverage this result to provide better task placement in data centers while taking into account other practical restrictions on task placement such as data locality?

- **Resilient Source Routing:** In order to strengthen the source routing control scheme further in the intra-data center environment, we need to tackle a few issues. Our solution has not dealt with response to switch and link failures. It is essential to build an auxiliary mechanism to route around failures, either with additional information in the header or through a network response mechanism.

- **Managing thousands of controllers in WAND:** Patronus assumes centralized control with global view. While our instantaneous scheduler can respond in seconds, several edge applications require response time of milliseconds. Even with a logically centralized controller, each MDC will require a local controller for detecting and responding to immediate local changes. Besides, the arbitrary interconnecting network topology calls for close inter-operation of network and server controllers. Realtime adaptation and self-organization with thousands of such controllers demand careful analysis of robustness and correctness. Several questions remain unanswered. How can we leverage the DCs' compute resources, which are considerable in the aggregate, to partition the optimization/control problem into local, distributed actions at thousands of sites along with some global guidance? In the data plane, what mechanisms are necessary to enable dynamic steering of traffic through a combination of MDCs and WAN?

- **Characterizing WAND service:** Cloud providers offer VMs with capabilities specified in terms of CPU, GPU, network, storage etc. In a WAND, defining the unit of service is cumbersome. A WAND provider will typically have macro- and micro-data centers with different compute capabilities and latency profiles between them. The cost and performance using the same amount of resource may vary widely depending on the size and location of the DCs and within the same DC across time due to limited statistical multiplexing. If WAND environment evolves towards resource offerings similar to cloud, service characterization should provide users with (a) cost of resources across multiple MDCs accessible to the user (which may vary with time), and (b) sufficient information to derive approximate latency performance in these MDCs with time. Alternatively, if the environment moves towards a serverless model, we need APIs that allow the user to specify their diverse needs (resource, bandwidth, latency constraints/location dependencies etc.) and novel control design which can translate these into resource allocations in a highly dynamic environment.

- **Interaction of multiple WANDs:** In future, applications may be spread across MDCs operated by multiple WAND providers. A multi-WAND environment adds further complexity to existing cloud-related problems such as migration, security, compliance etc. Multi-WAND deployment of applications may take various forms: across two carrier WANDs, across a the edge of a carrier and hyperscale DCs of content provider, in a WAND where the MDCs are operated by one provider and WAN by another, among others. In this environment, we need to ensure that interaction between various resource management schemes in different WANDs do not result in undesirable end-to-end performance for applications. How can we design WAND control schemes that can coordinate with external entities and make robust decisions using partial information?

- **Extracting the application intent for CadentFlow:** Determining deadlines/ priorities and optimization objective of applications can be cumbersome. However, system-level cues such as reads and writes to disk/memory has information for deriving the deadlines/priorities based on application behavior. This inspires design of learning-based metric estimation frameworks to learn and adapt the deadlines/priorities/weights based on application performance. Recent advances in flow prediction [140] are encouraging for the possibility of learning finer grained application characteristics. The application API may also be extended to allow the application developer to specify ANI preferences explicitly (as in TensorFlow). This is also feasible in serverless compute frameworks such as Lambda.

- **Intent-Awareness in Network Schedulers:** An intent-aware ANI opens avenue for new resource allocation schemes which can handle diverse objectives and constraints. Some of the challenges include ensuring fairness across applications with varied objectives and handling coexistence of CadentFlows with regular TCP connections. Scheduling under limited information is also an interesting problem to tackle. While our experiments focused on a single application with explicit knowledge of flow sizes and priorities, in practice, one or more of these factors may be unknown. This calls for mechanisms that can incorporate accuracy of metrics during scheduling and adapt in real-time to changes.

# Chapter 7: BIBLIOGRAPHY

[1] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 123–137, 2015.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264 pp. 10–10.

[3] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184 pp. 135–146.

[5] S. A. Jyothi, A. Singla, B. Godfrey, and A. Kolla, "Measuring and understanding throughput of network topologies," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 761–772.

[6] S. A. Jyothi, A. Singla, B. Godfrey, and A. Kolla, "Measuring and understanding throughput of network topologies," *CoRR*, vol. abs/1402.2531, 2014. [Online]. Available: http://arxiv.org/abs/1402.2531

[7] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-deployed Software Defined WAN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486019 pp. 3–14.

[8] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-driven WAN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486012 pp. 15–26.

[9] H. H. Liu, R. Viswanathan, M. Calder, A. Akella, R. Mahajan, J. Padhye, and M. Zhang, "Efficiently delivering online services over integrated infrastructure," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16).* Santa Clara, CA: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/liu pp. 77–90.

[10] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng, "Engineering egress with edge fabric: Steering oceans of content to the world," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098853 pp. 418–431.

[11] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat, "Taking the edge off with espresso: Scale, reliability and programmability for global internet peering," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098854 pp. 432–445.

[12] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 59–62, Jan. 2006. [Online]. Available: http://doi.acm.org/10.1145/1111322.1111336

[13] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XI. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2390231.2390237 pp. 31–36.

[14] S. A. Jyothi, M. Dong, and P. B. Godfrey, "Towards a flexible data center fabric with source routing," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2774993.2775005 pp. 10:1–10:8.

[15] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," in *SIGCOMM*, 2009.

[17] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a high performance, server-centric network architecture for modular data centers," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 63–74, Aug. 2009.

[18] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, “Portland: A scalable fault-tolerant layer 2 data center network fabric,” in *SIGCOMM*, 2009.

[19] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, “DCell: A scalable and fault-tolerant network structure for data centers,” in *SIGCOMM*, 2008.

[20] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan, “c-Through: Part-time Optics in Data Centers,” in *SIGCOMM*, 2010.

[21] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, “Helios: A hybrid electrical/optical switch architecture for modular data centers,” in *SIGCOMM*, 2010.

[22] A. Singla, A. Singh, K. Ramachandran, L. Xu, and Y. Zhang, “Proteus: a topology malleable data center network,” in *HotNets*, 2010.

[23] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, “Jellyfish: Networking data centers randomly,” in *NSDI*, 2012.

[24] A. R. Curtis, S. Keshav, and A. Lopez-Ortiz, “LEGUP: using heterogeneity to reduce the cost of data center network upgrades,” in *CoNEXT*, 2010.

[25] A. Curtis, T. Carpenter, M. Elsheikh, A. Lopez-Ortiz, and S. Keshav, “REWIRE: An optimization-based framework for unstructured data center network design,” in *INFOCOM, 2012 Proceedings IEEE*, March 2012, pp. 1116–1124.

[26] A. Valadarsky, M. Dinitz, and M. Schapira, “Xpander: Unveiling the secrets of high-performance datacenters,” in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIV. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2834050.2834059 pp. 16:1–16:7.

[27] F. Shahrokhi and D. Matula, “The maximum concurrent flow problem,” *Journal of the ACM*, vol. 37, no. 2, pp. 318–334, 1990.

[28] D. Padua, *Encyclopedia of Parallel Computing*, ser. Springer reference. Springer, 2011, no. v. 4, see bisection bandwidth discussion on p. 974. [Online]. Available: http://books.google.com/books?id=Hm6LaufVKFEC

[29] S. Chawla, R. Krauthgamer, R. Kumar, Y. Rabani, and D. Sivakumar, “On the hardness of approximating multicut and sparsest-cut,” *computational complexity*, 2006.

[30] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979.

[31] L. R. Ford and D. R. Fulkerson, “Maximal flow through a network,” *Canadian Journal of Mathematics*, vol. 8, pp. 399 –404, 1956.

[32] P. Elias, A. Feinstein, and C. Shannon, “A note on the maximum flow through a network,” *Information Theory, IEEE Transactions on*, vol. 2, no. 4, pp. 117–119, Dec 1956.

[33] T. Leighton and S. Rao, “Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms,” *J. ACM*, vol. 46, no. 6, pp. 787–832, Nov. 1999.

[34] A. Singla, P. B. Godfrey, and A. Kolla, “High throughput data center topology design,” in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2014.

[35] C. Chekuri, “Routing and network design with robustness to changing or uncertain traffic demands,” *SIGACT News*, vol. 38, no. 3, pp. 106–129, Sep. 2007.

[36] M. Kodialam, T. V. Lakshman, and S. Sengupta, “Traffic-oblivious routing in the hose model,” *IEEE/ACM Trans. Netw.*, vol. 19, no. 3, pp. 774–787, 2011.

[37] X. Yuan, S. Mahapatra, W. Nienaber, S. Pakin, and M. Lang, “A New Routing Scheme for Jellyfish and Its Performance with HPC Workloads,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’13, 2013, pp. 36:1–36:11.

[38] Gurobi Optimization Inc., “Gurobi optimizer reference manual,” http://www.gurobi.com, 2013.

[39] J. Kim, W. J. Dally, S. Scott, and D. Abts, “Technology-Driven, Highly-Scalable Dragonfly Topology,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA ’08, 2008, pp. 77–88.

[40] C. E. Leiserson, “Fat-trees: universal networks for hardware-efficient supercomputing,” *IEEE Trans. Comput.*, vol. 34, no. 10, pp. 892–901, Oct. 1985.

[41] J. Kim, W. J. Dally, and D. Abts, “Flattened butterfly: a cost-efficient topology for high-radix networks,” *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 126–137, June 2007.

[42] L. N. Bhuyan and D. P. Agrawal, “Generalized hypercube and hyperbus structures for a computer network,” *IEEE Tran. on Computers*, 1984.

[43] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, “Hyperx: Topology, routing, and packaging of efficient large-scale networks,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC ’09, 2009.

[44] R. V. Tomic, “Optimal networks from error correcting codes,” in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2013.

[45] M. Besta and T. Hoefler, “Slim Fly: A Cost Effective Low-Diameter Network Topology,” in *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC14)*, Nov. 2014.

[46] “Topology evaluation tool,” www.github.com/netarch/topobench.

[47] S. A. Jyothi, A. Singla, B. Godfrey, and A. Kolla, “Measuring and understanding throughput of network topologies,” http://arxiv.org/abs/1402.2531, 2014.

[48] B. Raghavan, M. Casado, T. Koponen, S. Ratnasamy, A. Ghodsi, and S. Shenker, “Software-defined internet architecture: Decoupling architecture from infrastructure,” in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XI, 2012, pp. 43–48.

[49] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, “Dynamic scheduling of network updates,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM ’14, 2014, pp. 539–550.

[50] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, “Conga: Distributed congestion-aware load balancing for datacenters,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM ’14, 2014, pp. 503–514.

[51] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca, “Planck: Millisecond-scale monitoring and control for commodity networks,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM ’14, 2014, pp. 407–418.

[52] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, “Fastpass: A centralized ”zero-queue” datacenter network,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM ’14, 2014, pp. 307–318.

[53] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, and C. Guo, “Explicit path control in commodity data centers: Design and applications,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, May 2015.

[54] R. Ramos, M. Martinello, and C. Esteve Rothenberg, “Slickflow: Resilient source routing in data center networks unlocked by openflow,” in *Local Computer Networks (LCN), 2013 IEEE 38th Conference on*, Oct 2013, pp. 606–613.

[55] C. Kim, M. Caesar, and J. Rexford, “Seattle: A scalable ethernet architecture for large enterprises,” *ACM Trans. Comput. Syst.*, vol. 29, no. 1, Feb. 2011.

[56] M. Yu, A. Fabrikant, and J. Rexford, “Buffalo: Bloom filter forwarding architecture for large organizations,” in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT ’09, 2009, pp. 313–324.

[57] K. Agarwal, C. Dixon, E. Rozner, and J. Carter, "Shadow MACs: Scalable Label-switching for Commodity Ethernet," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, 2014, pp. 157–162.

[58] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang, "Secondnet: A data center network virtualization architecture with bandwidth guarantees," in *Proceedings of the 6th International COnference*, ser. Co-NEXT '10. ACM, 2010.

[59] A. Schwabe and H. Karl, "Using MAC Addresses As Efficient Routing Labels in Data Centers," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14, 2014, pp. 115–120.

[60] M. Soliman, B. Nandy, I. Lambadaris, and P. Ashwood-Smith, "Source routed forwarding with software defined control, considerations and implications," in *Proceedings of the 2012 ACM Conference on CoNEXT Student Workshop*, ser. CoNEXT Student '12, 2012, pp. 43–44.

[61] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM.* ACM, 2012.

[62] E. Al-Shaer and S. Al-Haj, "FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration.* ACM, 2010.

[63] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with Anteater," in *ACM SIGCOMM*, August 2011.

[64] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static checking for networks," in *USENIX NSDI*, 2012.

[65] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013, pp. 15–28.

[66] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis." in *USENIX NSDI*, 2013.

[67] V. Sharma and F. Hellstrand, "Framework for multi-protocol label switching MPLS-based recovery," Internet Requests for Comments, RFC Editor, RFC 3469, February 2003.

[68] "Arista 7250QX data sheet," http://www.arista.com/assets/data/pdf/Datasheets/7250QX-64_Datasheet.pdf.

[69] "Cisco nexus 3000 series data sheet," http://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/white_paper_c11-713535.html.

[70] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang, "C3: Internet-scale control plane for video quality optimization," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ganjam pp. 131–144.

[71] AT&T, "The Cloud Comes to You: ATT to Power Self-Driving Cars, AR/VR and Other Future 5G Applications Through Edge Computing," http://bit.ly/attEdge, 2017.

[72] AT&T, "ATT Connected Car," http://bit.ly/attConn, 2018.

[73] "ECOMP (Enhanced Control, Orchestration, Management & Policy) Architecture White Paper," https://goo.gl/KUmNq2.

[74] "SDN-NFV Reference Architecture, Verizon Network Infrastructure Planning," https://goo.gl/YCeDBr.

[75] CORD, "Central Office Re-architected as a Datacenter (CORD)," https://goo.gl/qzzChH, 2016.

[76] D. Applegate and E. Cohen, "Making Intra-domain Routing Robust to Changing and Uncertain Traffic Demands: Understanding Fundamental Tradeoffs," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/863955.863991 pp. 313–324.

[77] Y. Li, J. Harms, and R. Holte, "A simple method for balancing network utilization and quality of routing," in *Proceedings. 14th International Conference on Computer Communications and Networks, 2005. ICCCN 2005.*, Oct 2005, pp. 71–76.

[78] S. Kandula, D. Katabi, B. Davie, and A. Charny, "Walking the tightrope: Responsive yet stable traffic engineering," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '05, 2005, pp. 253–264.

[79] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg, "COPE: Traffic Engineering in Dynamic Networks," in *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1159913.1159926 pp. 99–110.

[80] A. Elwalid, C. Jin, S. Low, and I. Widjaja, "Mate: Mpls adaptive traffic engineering," in *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 3, 2001, pp. 1300–1309 vol.3.

[81] N. Michael and A. Tang, “HALO: Hop-by-Hop Adaptive Link-State Optimal Routing,” *IEEE/ACM Transactions on Networking*, vol. 23, no. 6, pp. 1862–1875, Dec 2015.

[82] J. McCauley, Z. Liu, A. Panda, T. Koponen, B. Raghavan, J. Rexford, and S. Shenker, “Recursive sdn for carrier networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 4, pp. 1–7, Dec. 2016. [Online]. Available: http://doi.acm.org/10.1145/3027947.3027948

[83] A. A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, “Elasticon: An elastic distributed sdn controller,” in *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS ’14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2658260.2658261 pp. 17–28.

[84] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, “Logically centralized?: State distribution trade-offs in software defined networks,” in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN ’12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342443 pp. 1–6.

[85] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler, “Apache hadoop yarn: Yet another resource negotiator,” in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC ’13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633 pp. 5:1–5:16.

[86] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488 pp. 295–308.

[87] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, “Sparrow: Distributed, low latency scheduling,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522716 pp. 69–84.

[88] X. Meng, V. Pappas, and L. Zhang, “Improving the scalability of data center networks with traffic-aware virtual machine placement,” in *2010 Proceedings IEEE INFOCOM*, March 2010, pp. 1–9.

[89] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, “E2: A framework for nfv applications,” in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP ’15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815423 pp. 121–136.

[90] Z. Ayyub Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker, "A High Performance Packet Core for Next Generation Cellular Networks," in *Proceedings of the 2017 ACM SIGCOMM Conference*, 2017.

[91] Z. A. Qazi, P. K. Penumarthi, V. Sekar, V. Gopalakrishnan, K. Joshi, and S. R. Das, "KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2890955.2890961 pp. 2:1–2:12.

[92] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787505 pp. 421–434.

[93] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, and G. Varghese, "Wanalytics: Geo-distributed analytics for a data intensive world," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2723372.2735365 pp. 1087–1092.

[94] Wanghong Yuan, K. Nahrstedt, S. V. Adve, D. L. Jones, and R. H. Kravets, "Grace-1: cross-layer adaptation for multimedia quality and battery energy," *IEEE Transactions on Mobile Computing*, vol. 5, no. 7, pp. 799–815, July 2006.

[95] W. Yuan and K. Nahrstedt, "Energy-efficient cpu scheduling for multimedia applications," *ACM Trans. Comput. Syst.*, vol. 24, no. 3, pp. 292–331, Aug. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151690.1151693

[96] Wanghong Yuan, K. Nahrstedt, and Kihun Kim, "R-edf: a reservation-based edf scheduling algorithm for multiple multimedia task classes," in *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, May 2001, pp. 149–154.

[97] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time cpu scheduling for mobile multimedia systems," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/945445.945460 pp. 149–163.

[98] B. Li, D. Xu, K. Nahrstedt, and J. Liu, "End-to-end qos support for adaptive applications over the internet," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 3529, pp. 166–176, 12 1998.

[99] V. IO, "Vapor IO Chamber," https://www.vapor.io/chamber/, 2015.

[100] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 377–392.

[101] I. Markit, “North American security camera installed base to reach 62 million in 2016,” http://bit.ly/cctvInUS, 2016.

[102] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, “Making sense of performance in data analytics frameworks,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout pp. 293–307.

[103] “Telecommunications Database,” https://www.telcodata.us, 2018.

[104] D. World, “U.S. Population by zip code, 2010-2016,” http://bit.ly/usPop, 2016.

[105] R. Durairajan, P. Barford, J. Sommers, and W. Willinger, “intertubes: A study of the us long-haul fiber-optic infrastructure.”

[106] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, “Managing data transfers in computer clusters with orchestra,” in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM ’11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018448 pp. 98–109.

[107] M. Chowdhury and I. Stoica, “Efficient coflow scheduling without prior knowledge,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM ’15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787480 pp. 393–406.

[108] M. Chowdhury, Y. Zhong, and I. Stoica, “Efficient coflow scheduling with varys,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM ’14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626315 pp. 443–454.

[109] L. Chen, W. Cui, B. Li, and B. Li, “Optimizing coflow completion times with utility max-min fairness,” in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, April 2016, pp. 1–9.

[110] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, “Sincronia: Near-optimal network design for coflows,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3230543.3230569 pp. 16–29.

[111] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, “Better never than late: Meeting deadlines in datacenter networks,” in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM ’11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018443 pp. 50–61.

[112] C.-Y. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2342356.2342389 pp. 127–138.

[113] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934880 pp. 160–173.

[114] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "TensorFlow: A System for Large-Scale Machine Learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[115] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration," 2017.

[116] X. S. Wang, A. Krishnamurthy, and D. Wetherall, "Speeding up web page loads with shandian," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/wang pp. 109–122.

[117] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan, "Watchtower: Fast, secure mobile page loads using remote dependency resolution," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '19. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3307334.3326104 pp. 430–443.

[118] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems.* ACM, 2015, p. 1.

[119] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.* ACM, 2010, pp. 135–146.

[120] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[121] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2960414.2960419 pp. 5:1–5:6.

[122] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: A graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592799 pp. 1:1–1:14.

[123] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles.* ACM, 2013, pp. 439–455.

[124] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* ACM, 2015, pp. 239–250.

[125] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[126] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham et al., "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data.* ACM, 2014, pp. 147–156.

[127] Z. Yang, W. Wu, K. Nahrstedt, G. Kurillo, and R. Bajcsy, "Enabling multi-party 3d tele-immersive environments with viewcast," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 6, no. 2, pp. 7:1–7:30, Mar. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1671962.1671963

[128] A. Arefin, R. Rivas, and K. Nahrstedt, "Osm: Prioritized evolutionary qos optimization for interactive 3d teleimmersion," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 10, no. 1s, pp. 12:1–12:24, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2543899

[129] A. Arefin, Z. Huang, K. Nahrstedt, and P. Agarwal, "4d telecast: Towards large scale multi-site and multi-view dissemination of 3dti contents," in *2012 IEEE 32nd International Conference on Distributed Computing Systems*, June 2012, pp. 82–91.

[130] A. Arefin, R. Rivas, R. Tabassum, and K. Nahrstedt, "Opensession: Sdn-based cross-layer multi-stream management protocol for 3d teleimmersion," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, Oct 2013, pp. 1–10.

[131] A. Arefin and K. Nahrstedt, "Multi-stream frame rate guarantee using cross-layer synergy," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, Oct 2013, pp. 1–2.

[132] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009. [Online]. Available: https://doi.org/10.14778/1687553.1687609

[133] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855732 pp. 21–21.

[134] J. Wang, H. Zhou, Y. Hu, C. d. Laat, and Z. Zhao, "Deadline-aware coflow scheduling in a dag," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2017, pp. 341–346.

[135] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[136] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. [Online]. Available: http://arxiv.org/abs/1409.4842

[137] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. [Online]. Available: http://arxiv.org/abs/1512.00567

[138] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[139] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[140] V. Dukic, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla, "Is advance knowledge of flow sizes a plausible assumption?" in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019.

[141] M. Babaioff and J. Chuang, "On the optimality and interconnection of valiant load-balancing networks," in *INFOCOM*, 2007.

[142] F. Chung, "Laplacians of graphs and cheeger's inequalities," in *Combinatorics, Paul Erdös is Eighty, Vol. 2*. Janos Bolyai Mathematical Society, Budapest, 1996, pp. 157–172.

## Appendix A: PROOF OF THEOREM 2.1

Appendix A.1: We revisit the *maximum concurrent flow* problem, based on which we defined throughput in §2.1: Given a network $G = (V, E_G)$ with capacities $c(u, v)$ for every edge $(u, v) \in E_G$, and a collection (not necessarily disjoint) of pairs $(s_i, t_i), i = 1, \ldots, k$ each having a unit flow demand, we are interested in maximizing the minimum flow. Instead of the traffic matrix (TM) formulation of §2.1, for the following discussion, it will be convenient to think of the pairs of vertices that require flow between them as defining a demand graph, $H = (V, E_H)$. Thus, given $G$ and $H$, we want the maximum throughput. As we noted in §2.1, this problem can be formulated as a standard linear program, and is thus computable in polynomial time.

We are interested in comparing our suggested throughput metric with sparsest cut. We first prove the following theorem.

**Theorem A.1:** The dual of the linear program for computing throughput is a linear programming relaxation for sparsest cut.

*Proof.* We shall use a formulation of the throughput linear program that involves an exponential number of variables but for which is easier to derive the dual. We denote by $P_{s,t}$ the set of all paths from $s$ to $t$ in $G$ and we introduce a variable $x_p$ for each path $p \in P_{s,t}$, for each $(s, t) \in E_H$, corresponding to how many units of flow from $s$ to $t$ are routed through path $p$.

$$\begin{aligned}
&\max && y && && \text{(A.1)}\\
&\text{subject to} && \sum_{p \in P_{s,t}} x_p \geq y && \forall (s,t) \in E_H, && \text{(A.2)}\\
& && \sum_{p:(u,v) \in p} x_p \leq c(u,v) && \forall (u,v) \in E_G && \text{(A.3)}\\
& && x_p \geq 0 && \forall p && \text{(A.4)}\\
& && y \geq 0. && && \text{(A.5)}
\end{aligned}$$

The dual of the above linear program will have one variable $w(s, t)$ for each $(s, t) \in E_H$ and one variable $z(u, v)$ for each $(u, v) \in E_G$.

$$\begin{aligned}
\text{min} \quad & \sum_{u,v} z(u,v)c(u,v) && && \text{(A.6)}\\
\text{subject to} \quad & \sum_{(s,t)\in E_H} w(s,t) \geq 1 && && \text{(A.7)}\\
& \sum_{(u,v)\in p} z(u,v) \geq w(s,t) && \forall (s,t) \in E_H, p \in P_{s,t} && \text{(A.8)}\\
& w(s,t) \geq 0 && \forall (s,t) \in E_H && \text{(A.9)}\\
& z(u,v) \geq 0 && \forall (u,v) \in E_G. && \text{(A.10)}
\end{aligned}$$

It is not hard to realize that in an optimal solution, without loss of generality, $w(s,t)$ is the length of the shortest path from $s$ to $t$ in the graph weighted by the $z(u,v)$. We can also observe that in an optimal solution we have $\sum w(s,t) = 1$. These remarks imply that the above dual is equivalent to the following program, where we introduce a variable $l(x,y)$ for every pair or vertices in $E_G \cup E_H$.

$$\begin{aligned}
\text{min} \quad & \sum_{u,v} l(u,v)c(u,v) && && \text{(A.11)}\\
\text{subject to} \quad & \sum_{(s,t)\in E_H} l(s,t) = 1 && && \text{(A.12)}\\
& \sum_{(u,v)\in p} l(u,v) \geq l(s,t) && \forall (s,t) \in E_H, p \in P_{s,t} && \text{(A.13)}\\
& l(u,v) \geq 0 && \forall (u,v) \in E_G \cup E_H && \text{(A.14)}\\
& && && \text{(A.15)}
\end{aligned}$$

The constraints $\sum_{(u,v)\in p} l(u,v) \geq l(s,t)$ can be equivalently restated as triangle inequalities. This means that we require $l(u,v)$ to be a metric over $V$. These observations give us one more alternative formulation:

$$\min_{l(\cdot,\cdot)\text{ metric}} \frac{\sum_{(u,v)\in E_G} c(u,v) \cdot l(u,v)}{\sum_{(s,t)\in E_H} l(s,t)} \tag{A.16}$$

We can finally see that the above formulation is a linear programming relaxation for a cut problem. More specifically, the sparsest cut problem is asking to find a cut $S$ that minimizes the ratio

$$\frac{\sum_{(u,v)\in E_G \text{ cut by S}} c(u,v)}{|\text{ edges} \in E_H \text{ cut by S}|} \tag{A.17}$$

This is equivalent to minimizing ratio (A.16) over $\ell_1$ metrics only.

If we take $E_H$ to be the complete graph (corresponding to all-to-all demands), we get the standard sparsest cut definition:

$$\frac{\sum_{(u,v)\in E_G \text{ cut by S}} c(u,v)}{|S||\bar{S}|} \tag{A.18}$$

Before we prove Theorem 1 from §2.1, we shall demonstrate the following claim.

**Claim:** If $G$ is a $d$-regular expander graph on $N$ nodes and $H$ is the complete graph, the value of the linear program for throughput is $\emptyset(\frac{d\log d}{N\log N})$. The value of the sparsest cut is $\Omega(\frac{d}{N})$.

*Proof.* Let us denote by $T$ the optimal value of expression (A.16). Note that this is the optimal value of the dual for the linear program for throughput, therefore equal to the optimal throughput. By taking $l(\cdot, cdot)$ to be the shortest path metric on $G$, we calculate:

$$T \leq \frac{\sum_{(i,j)\in E_G} l(i,j)}{\sum_{i,j\in V} l(i,j)} \leq \frac{d/2.|V|}{\Theta(N^2\frac{\log N}{\log d})} \leq O(\frac{d\log d}{N^2\log N}) \tag{A.19}$$

Here, the first inequality follows from the fact that for $d$-regular graphs, each node can reach no more than $d^i$ nodes in $i$ hops. This means that given a vertex $v$, there exist $\Theta(N)$ nodes with distance at least $\frac{\log N}{\log d}$ from it, which means that the total distance between all pairs of nodes is $\Theta(N^2\frac{\log N}{\log d})$.

Let $\Phi$ denote the minimum value of ratio (A.18) for $G$. Since $G$ is an expander, this ratio is

$$\Phi \geq \Omega(\frac{d\cdot|S|}{|S||V-S|}) = \Omega(\frac{d}{N}) \tag{A.20}$$

Restating Theorem 2.1 from Chapter 2:

**Theorem 2.1.** *Let graph $G$ be any $2d$-regular expander on $N = \frac{n}{dp}$ nodes, where $d$ is a constant and $p$ is a free parameter. Let graph $B$ be constructed by replacing each edge of $G$ with a path of length $p$. Then, $B$ has throughput $T_B = O(\frac{1}{np\log n})$ and sparsest cut $\Phi_B = \Omega(\frac{1}{np})$.*

*Proof.* Let $(S_1, S_2)$ be the sparsest cut in $B$. Let $({S_1}', {S_2}')$ be the corresponding cut in $G$. Namely, if an edge was cut in $B$ by $(S_1, S_2)$ that belonged to a path $p_e$ then $({S_1}', {S_2}')$ cuts $e$. Let $\Phi_B$ be the value of the cut $(S_1, S_2)$ in $A$ and $\Phi_G$ the value of $({S_1}', {S_2}')$ in $G$. Then

$$\Phi_B = \frac{E(S_1, S_2)}{|S_1||S_2|} = \frac{E({S_1}', {S_2}')}{|S_1||S_2|} \geq \frac{E({S_1}', {S_2}')}{p \cdot |S_1'|p \cdot |S_2'|} \geq \frac{\Phi_G}{p^2}$$

by equation (A.20) we have $\Phi_G \geq \Omega(\frac{1}{N}) = \Omega(\frac{p}{n})$ which gives us

$$\Phi_B \geq \Omega(\frac{1}{np})$$

On the other hand, let $T_B$ be the value of the throughput of $B$. We follow a similar reasoning as we did in equation (A.19).

$$\begin{aligned} T_B \leq \frac{\sum_{(i,j)\in E_G} l(i,j)}{\sum_{i,j\in V} l(i,j)} &\leq \frac{Ndp}{\Theta((Np)^2 p \log N)} \\ &\leq O(\frac{1}{Np^2 \log N}) = O(\frac{1}{np\log n}) \end{aligned} \tag{A.21}$$

## Appendix B: PROOF OF THEOREM 2.2

Appendix B.1:

*Proof.* $T_{A2A}$ has demand $\frac{1}{n}$ on each flow, so the largest feasible multicommodity flow routing of $T_{A2A}$ in $G$ has capacity $\frac{t}{n}$ allocated to each flow. Let $C$ be a graph representing this routing, i.e., a complete digraph with capacity $\frac{t}{n}$ on each link. Systems-oriented readers may find it useful to think of $C$ as an overlay network implemented with reserved bandwidth in $G$. In other words, to prove the theorem, it is sufficient to show that taking $T$ to be any hose-model traffic matrix, $T \cdot t/2$ is feasible in $C$.

For this, we use a two-hop routing scheme analogous to Valiant load balancing [141]. Consider any traffic demand $vw$. In the first step, we split this demand flow into $n$ equal parts, routing flow $\frac{1}{n} \cdot T(v, w) \cdot t/2$ from $v$ to every node in the network, along the direct links (or the zero-hop path when the target is $v$ itself). In the second step, the traffic arriving at each node is sent along at most one link to its final destination.

We now have to show that this routing is feasible in $C$. Consider any link $i \to j$. This link will carry a fraction $\frac{1}{n}$ of all the traffic originated by $i$, and a fraction $\frac{1}{n}$ of all the traffic destined to $j$. Because $T$ is a hose model TM, each node originates and sinks a total of $\leq 1$ unit of traffic; and since we are actually attempting to route the scaled traffic matrix $T \cdot t/2$, each node originates and sinks a total of $\leq t/2$ units of traffic. Therefore, link $i$ carries a total of

$$\frac{t}{2} \cdot \frac{1}{n} + \frac{t}{2} \cdot \frac{1}{n} = \frac{t}{n},$$

which is the available capacity on each link of $C$ and is hence feasible.

## Appendix C: MEASURING CUTS

Appendix C.1: We employ several heuristics for estimating sparsest cut.

**Brute-force computation** Brute force computation of sparsest cut is computationally intensive since it considers all possible cuts in the network ($2^{n-1}$ cuts in a network with $n$ nodes). In addition to bandwidth, the number of flows traversing each cut has to be estimated which adds further overhead in the computation of sparsest cut.

Due to the computational complexity, brute force evaluation of sparsest cut is possible only for networks of size less than 20. However, we perform limited brute-force computation on all networks by capping the computation at $10,000$ cuts.

**One-node cuts** Designed computer networks as well as naturally occurring networks tend to be denser at the core and sparse at the edges. When the core has high capacity, it is likely that the worst-case cut occurs at the edges. Hence, this heuristic considers all cuts with only a single node in a subset formed by the cut. There exists $n$ cuts with a single node. This is a very small fraction of the total $2^{n-1}$ cuts.

**Two-node cuts** $\frac{n*(n-1)}{2}$ cuts with two nodes in a subset also reveal the limited connectivity at the edges of the network.

**Expanding cuts** It is likely that the network is clustered,i.e., it contains two or more highly connected components connected by a few links. Subsets of all possible combinations of contiguous nodes in the network can be very large. We optimize our search to a subspace of this category of cuts. Starting from each node, we consider cuts which include nodes within a distance $k$ from the node. When $k = 0$, the cut involves only the originating node and is equivalent to the single node case discussed before. When $k = 1$, all nodes within distance 1 from the node are considered – the node and its neighbors. $k$ is incremented until the entire graph is covered. If $d$ is the diameter of the network, the number of cuts considered is less than or equal to $n * d$.

| Topology family | Total | #networks with throughput= estimated cut | Sparsest cut estimator which found the worst cut | | | | |
|---|---|---|---|---|---|---|---|
| | | | Brute force | 1-node | 2-node | Expanding regions | Eigenvector |
| BCube | 7 | 2 | 2 | 0 | 0 | 3 | 7 |
| DCell | 4 | 2 | 2 | 0 | 0 | 2 | 3 |
| Dragonfly | 4 | 0 | 2 | 0 | 0 | 0 | 2 |
| Fat tree | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| Flattened butterfly | 8 | 5 | 6 | 0 | 1 | 0 | 5 |
| Hypercube | 7 | 3 | 3 | 0 | 0 | 1 | 6 |
| HyperX | 11 | 1 | 1 | 0 | 0 | 1 | 10 |
| Jellyfish | 350 | 3 | 0 | 0 | 0 | 2 | 349 |
| LongHop | 110 | 9 | 45 | 0 | 0 | 1 | 66 |
| SlimFly | 6 | 1 | 1 | 0 | 0 | 0 | 5 |
| Natural networks | 66 | 48 | 18 | 21 | 11 | 34 | 38 |
| Total | 581 | 82 | 88 | 29 | 20 | 52 | 499 |

Table C.1: Estimated sparsest cuts: Do they match throughput, and which estimators produced those cuts?

**Eigenvector based optimizations** Eigenvector corresponding to the second smallest eigenvalue of the normalized Laplacian of a graph can give a set of $n$ cuts, the worst of which is within a constant factor from the actual cut [142]. The nodes of the graph are sorted in the ascending order corresponding to their value in the second eigenvector [142]. We sweep this vector of sorted nodes to obtain the $n$ cuts.

How well did our sparse cut heuristics perform? Comparing columns 2 and 3 in Table C.1, we see that cuts accurately predicted throughput in less than 15% of the tested networks only. Table C.1 shows how often each estimator found the sparse cut. More than one technique may find the sparse cut, hence the sum may not equal the total number of networks. Brute-force computation was helpful in finding 15% of the sparse cuts. Cuts involving one or two nodes and contiguous regions of the graph also found the sparse cut in a small fraction of networks (less than 10% each). The majority of such networks are the natural networks, which are often denser in the core and sparser in the edges. Sparse connectivity at the edges lead to bottlenecks at the edge which are revealed by cuts involving one or two nodes. Fat tree is another interesting case where every heuristic's cuts yield the accurate flow value. Overall, the eigenvector-based approximation found the largest number of sparse cuts (86%), but it is known not to be a tight approximation [142], and the full collection of heuristics did improve on it in a nontrivial fraction of cases.