OneSwitch Data Center Architecture

Wile Sehery

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

T. Charles Clancy, Chair

Lamine M Mili

Ing-Ray Chen

Thidapat Chantem

Ryan Gerdes

February 16, 2018

Arlington, Virginia

Keywords: Data Center, SDN, OpenFlow, Flow Optimization, Clos, Supermarket,

Flow-Commodity, Load Balancing

OneSwitch Data Center Architecture

Wile Sehery

ABSTRACT

In the last two-decades data center networks have evolved to become a key element in improving levels of productivity and competitiveness for different types of organizations. Traditionally data center networks have been constructed with 3 layers of switches, Edge, Aggregation, and Core. Although this Three-Tier architecture has worked well in the past, it poses a number of challenges for current and future data centers.

Data centers today have evolved to support dynamic resources such as virtual machines and storage volumes from any physical location within the data center. This has led to highly volatile and unpredictable traffic patterns. Also The emergence of "Big Data" applications that exchange large volumes of information have created large persistent flows that need to coexist with other traffic flows. The Three-Tier architecture and current routing schemes are no longer sufficient for achieving high bandwidth utilization.

Data center networks should be built in a way where they can adequately support virtualization and cloud computing technologies. Data center networks should provide services such as, simplified provisioning, workload mobility, dynamic routing and load balancing, equidistant bandwidth and latency. As data center networks have evolved the Three-Tier architecture has proven to be a challenge not only in terms of complexity and cost, but it also falls short of supporting many new data center applications.

In this work we propose OneSwitch: A switch architecture for the data center. OneSwitch is backward compatible with current Ethernet standards and uses an OpenFlow central controller, a Location Database, a DHCP Server, and a Routing Service to build an Ethernet fabric that appears as one switch to end devices. This allows the data center to use switches in scale-out topologies to support hosts in a plug and play manner as well as provide much needed services such as dynamic load balancing, intelligent routing, seamless mobility, equidistant bandwidth and latency.

OneSwitch Data Center Architecture

Wile Sehery

GENERAL AUDIENCE ABSTRACT

In the last two-decades data center networks have evolved to become a key element in improving levels of productivity and competitiveness for different types of organizations. Traditionally data center networks have been constructed with 3 layers of switches. This Three-Tier architecture has proven to be a challenge not only in terms of complexity and cost, but it also falls short of supporting many new data center applications.

In this work we propose OneSwitch: A switch architecture for the data center. OneSwitch supports virtualization and cloud computing technologies by providing services such as, simplified provisioning, workload mobility, dynamic routing and load balancing, equidistant bandwidth and latency.

Dedication

This is dedicated to my dear father and mother for their endless love and encouragement, to my amazing wife and wonderful son, for their patience and understanding. I could never have finished my graduate studies without all of your support.

Acknowledgments

I would like to thank my dissertation advisor, Dr. Charles Clancy, for his guidance and support throughout my graduate studies and for the opportunity to work under his supervision.

I would also like to thank the members of my committee, Dr. Lamine M Mili, Dr. Ing-Ray Chen, Dr. Tam Chantem, and Dr. Ryan Gerdes for their guidance and feedback.

Finally, I would like to thank the many people who have supported and motivated me during my research.

Contents

List of Figures **xii**

List of Tables **xviii**

1 Introduction **1**

1.1 Data Center Applications 2

1.2 Data Center Traffic 4

1.3 Data Center Network Architecture 6

1.4 Switching and Routing 8

1.5 Motivation 9

1.6 Objective 12

1.7 Contribution 13

1.8 Overview 15

2 Background and Related Work **16**

2.1 Related Work 19

2.2 Three-Stage Clos Switch 23

2.3 Data Centers with Folded Clos Networks 28

2.4 Load Balancing in Folded Clos Networks 29

2.5 ECMP and Related Work 31

2.6 Mininet 33

3 OneSwitch **36**

3.1 OneSwitch Building Blocks 38

3.1.1 OpenFlow 38

3.1.2 SDN Controller 42

3.1.3 Open vSwitch 46

3.2 OneSwitch Architecture 49

3.3 OneSwitch Packet Forwarding 53

3.3.1 Packet forwarding between hosts on the same edge switch 53

3.3.2 Packet forwarding between hosts on different edge switches 56

3.3.3 Packet forwarding between mobile hosts 59

3.4 OneSwitch Features 61

4 Load Balancing in OneSwitch **63**

4.1 Balls and Bins Model 64

4.2 The Power of Two Choices 67

4.3 Supermarket Model 68

4.4 Formulating load balancing in OneSwitch as a Supermarket Problem 70

4.5 Selective Randomized Load Balancing (SRL) 78

4.6 SRL Simulation 83

5 Flow Optimization in OneSwitch **87**

5.1 Multicommodity Flow Problem 88

5.1.1 Integer Multicommodity Flow Problem 88

5.1.2 Binary Multicommodity Flow Problem 89

5.2 Formulating routing in OneSwitch as a Binary Multicommodity Flow Problem 89

5.3 Optimal routing without flow collisions 95

5.4 Optimal routing with minimal flow collisions 115

5.4.1 Optimized Flow Re-routing (OFR) 115

5.4.2 FlowFit 118

5.4.3 Simulating routing with minimal flow collisions 121

5.4.4 FlowFit and OFR Design Considerations 124

6 OneSwitch Evaluation **128**

6.1 Control Plane Scalability 128

6.1.1 POX Controller 129

6.1.2 Location Database 140

6.2 Data Plane Scalability 142

6.2.1 Number of Flow Rules 143

6.2.2 Time to Search Flow Rules 144

6.3 Routing Service 145

6.4 Practicality of OneSwitch 155

6.5 OneSwitch compared to other schemes 158

6.5.1 Main Objective 159

6.5.2 Packet Forwarding 159

6.5.3 Size of Forwarding Tables 160

6.5.4 Load Balancing 161

6.5.5 Reaction to Congestion 163

6.5.6 Targeted Flows 165

6.5.7 Data Plane Topology 165

6.5.8 Control Plane 166

6.5.9 Fault Tolerance 166

6.5.10 Mobility 167

6.5.11 Implementation 167

6.5.12 Packet Delivery 168

7 Conclusion and Future Work **174**

7.1 Conclusion 174

7.2 Future Work 178

Bibliography 179

List of Figures

1.1 A survey of the interconnect choices in the Top 500 supercomputer rankings. 2

1.2 A modern data center. 3

1.3 Data center traffic measurements published by Microsoft. 5

1.4 Three-tier network model. 7

2.1 Switch-Centric networks (a) Hypercube. (b) Flattened Butterfly. (c) DragonFly. 23

2.2 Hybrid networks (a) FlatNet. (b) DCell. (c) BCube. 23

2.3 Single Stage switch. 24

2.4 Three-Stage Switch Topology. 25

2.5 Data center network with a Folded Clos topology. 29

2.6 2x4 Folded Clos Network. 31

2.7 Mininet Editor. 34

3.1 Legacy network vs SDN. 37

3.2 Software-Defined Networking (SDN) framework. 38

3.3 OpenFlow Switch Architecture. 40

3.4 OpenFlow Counters. 41

3.5 OpenFlow Controllers. 43

3.6 Open vSwitch. 46

3.7 Open vSwitch internal architecture. 47

3.8 OneSwitch Architecture. 50

3.9 Packet forwarding between hosts on the same edge switch. 55

3.10 Packet forwarding between hosts on different edge switches. 57

3.11 Packet forwarding between hosts on different edge switches. 60

4.1 Average time vs number of queried uplinks for $\lambda = 0.55$ and $L = 0.01, 0.03, 0.05$ and 0.1. 75

4.2 Average time vs number of queried uplinks for $\lambda = 0.65$ and $L = 0.01, 0.03, 0.05$ and 0.1. 75

4.3 Average time vs number of queried uplinks for $\lambda = 0.75$ and $L = 0.01, 0.03, 0.05$ and 0.1. 76

4.4 Average time vs number of queried uplinks for $\lambda = 0.75$ and $L = 0.01, 0.03$ and 0.05. 76

4.5 104-bit 5-Tuple Flow ID. 78

4.6 SRL link selection. 80

4.7 Normalized hash collision-Step/Uniform. 85

4.8 Normalized hash collisions-Shuffle/Predominately mice. 86

4.9 Normalized hash Collisions-Random/Predominately elephant. 86

5.1 OneSwitch network represented as a directed graph. 90

5.2 Number of decision variables versus the total number of switches for 1000 flows. 93

5.3 Simultaneous Flow Routing. 94

5.4 Weighted graph coloring in OneSwitch. 95

5.5 OneSwitch dataplane represented as a directed graph. 96

5.6 The structure of the A matrix. 99

5.7 Link utilization for uniform flows and step traffic pattern. 103

5.8 Link utilization for predominately mice flows and step traffic pattern. 104

5.9 Link utilization for predominately elephant flows and step traffic pattern. . . 105

5.10 Runtime vs Number of flows for different flow types and step traffic pattern. 106

5.11 Link utilization for uniform flows and shuffle traffic pattern. 107

5.12 Link utilization for predominately mice flows and shuffle traffic pattern. . . . 108

5.13 Link utilization for predominately elephant flows and shuffle traffic pattern. . 109

5.14 Runtime vs number of flows for different flow types and shuffle traffic pattern. 110

5.15 Link utilization for uniform flows and random traffic pattern. 111

5.16 Link utilization for predominately mice flows and random traffic pattern. . . 112

5.17 Link utilization for predominately elephant flows and random traffic pattern. 113

5.18 Runtime vs number of flows for different flow types and random traffic pattern.114

5.19 Elephant flow re-routing in FlowFit. 121

5.20 Normalized flow collisions-Predominately mice. 123

5.21 Normalized flow collisions-Predominately elephant. 123

5.22 Normalized flow collisions-Uniform. 124

6.1 Cbench sample output. 133

6.2 Throughput vs number of switches. 134

6.3 ofdump output. 135

6.4 ofstats output. 135

6.5 Max response time vs number of switches. 136

6.6 Polling frequency vs response time. 137

6.7 Polling frequency vs throughput. 138

6.8 Flows installed in switch vs response time. 139

6.9 Flows installed in switch vs throughput 140

6.11 Response time vs number of switches. 141

6.10 Throughput vs number of switches. 142

6.12 Flow Rules 143

6.13 Latency due to searching flow rules. 145

6.14 K=48, H=512, F=8, and T=Step. 147

6.15 K=48, H=512, F=8, and T=Shuffle. 148

6.16 K=48, H=512, F=8, and T=Random. 148

6.17 K=48, H=512, F=8, and T=Random. 150

6.18 K=48, H=512, F=8, and T=Step. 150

6.19 K=48, H=512, F=8, and T=Shuffle. 151

6.20 Normalized FCT for elephant flows, K=48, H=512, and F=4. 152

6.21 Normalized FCT for elephant flows, K=48, H=512, and F=8. 152

6.22 Normalized FCT for mice flows, K=48, H=512, and F=4. 153

6.23 Normalized FCT mice flows, K=48, H=512, and F=8. 153

6.24 Data Center measurements published by Microsoft Research 155

6.25 Inter-Arrival Time 157

6.26 Average FCT for SRL, MPTCP, and Flowlet. 162

6.27 Average FCT for FlowFit, Hedera, and Presto. 164

7.1 Controller partitioning. 177

7.2 Controller parallelism. 177

List of Tables

4.1 d and $T_d(\lambda)$ for different values of λ and L 77

4.2 SRL d lookup table 81

6.1 OneSwitch vs other schemes. 171

Chapter 1

Introduction

A decade ago high-performance networks were mostly custom and proprietary. Today, however, the most powerful computers are increasingly using Ethernet, according to a survey of interconnect choices in the Top 500 supercomputer rankings, shown in Figure 1.1.

Ethernet networks have bridged the performance and scalability gap between system architecture built using commodity-off-the-shelf components and purpose-built custom system architectures.

The proliferation of the Ethernet standard combined with economies of scale provide the basic building block of a modern data center network. A modern data center, shown in Figure 1.2, is home to tens of thousands of hosts each consisting of processors, memory, network interface cards, and high-speed I/O disks or flash. Large number of hosts are grouped together and placed in pods of compute resources that are tightly connected with

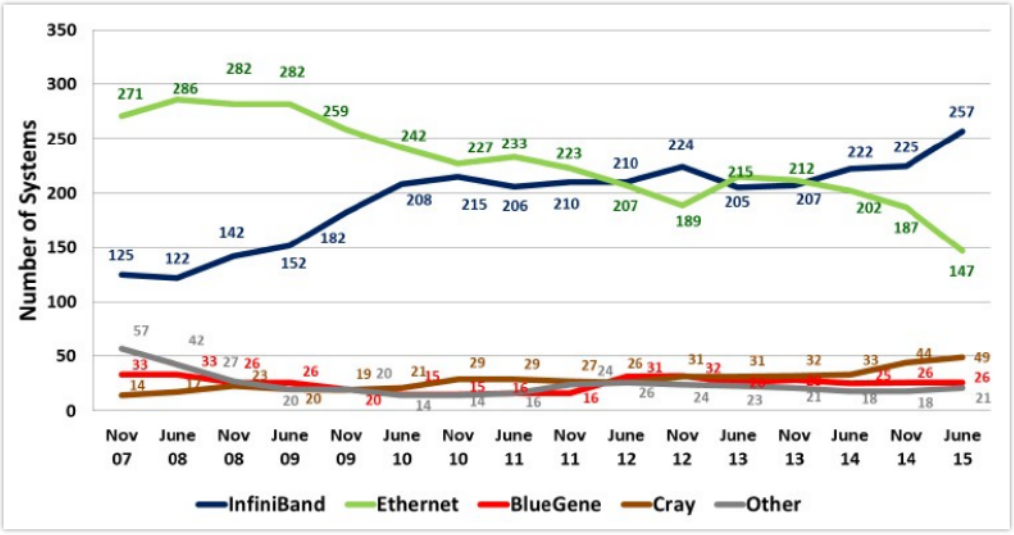

Figure 1.1: A survey of the interconnect choices in the Top 500 supercomputer rankings.

a high-bandwidth network.

1.1 Data Center Applications

Modern data centers support applications that have drastically varying requirements. On one hand, Web services are increasingly structured as a set of hierarchical components that must pass a series of small, inter-dependent communication messages between them in order to construct a response to incoming requests. On the other hand, "Big Data" applications systems like Spark [96] and TritonSort [76] rely on shuffling large amounts of state from each node to every other node. The overall throughput of these so-called Partition/Aggregate workloads [54] heavily relies on the latency of the slowest communication message.

Similarly, structured stores like BigTable [34] and their front-ends, such as Memcached [111]

Figure 1.2: A modern data center.

require highly parallel access to a large number of content nodes to persist state across a number of machines, or to reconstruct state that is distributed through the data center. In these latter cases, low-latency access between clients and their servers is critical for good application performance.

To reduce the likelihood of congestion, the network can be over provisioned by providing an abundance of bandwidth for different traffic patterns. Over provisioning within large-scale networks is prohibitively expensive. One solution is to implement quality of service polices to segregate traffic into distinct classes and provide differentiated service in order to meet application-level SLA's. The end goal is a high performance network that provides predictable latency and bandwidth characteristics across varying traffic patterns.

1.2 Data Center Traffic

Traffic in data center networks is usually asymmetric with client-to-server requests being many in number but generally small in size. On the other hand Server-to-client responses are few in number but large in size. The traffic is often measured and characterized according to flows, which are sequences of packets from a source to a destination host. When referring to Internet protocols, a flow is further refined to include a specific source and destination port number and transport type, UDP or TCP, for example.

Data center traffic is highly aggregated, and as a result the average of traffic flows says very little about the actual flow sizes. This is due to the high degree of variability between flows being aggregated. As a result, even a network that is not heavily utilized can experience lots of packet discards.

To understand individual flow characteristics better flows are sometimes categorized into classes. The most common classification is using the so-called "Elephant" and "Mice" classes. Elephant flows are relatively low in number but account for a high percentage of the traffic volume. They have a large number of packets and are generally long lived. Mice flows are many in number but account for a small percentage of traffic and generally are short lived.

Figure 1.3 shows data center traffic measurements published by Microsoft. As can be seen most traffic flows are mice, combined they consume less bandwidth than a much smaller number of elephant flows with duration's ranging from 10 seconds to 100 seconds.

Elephant flows can cause "hotspot" links that can lead to congestion or discarded packets.

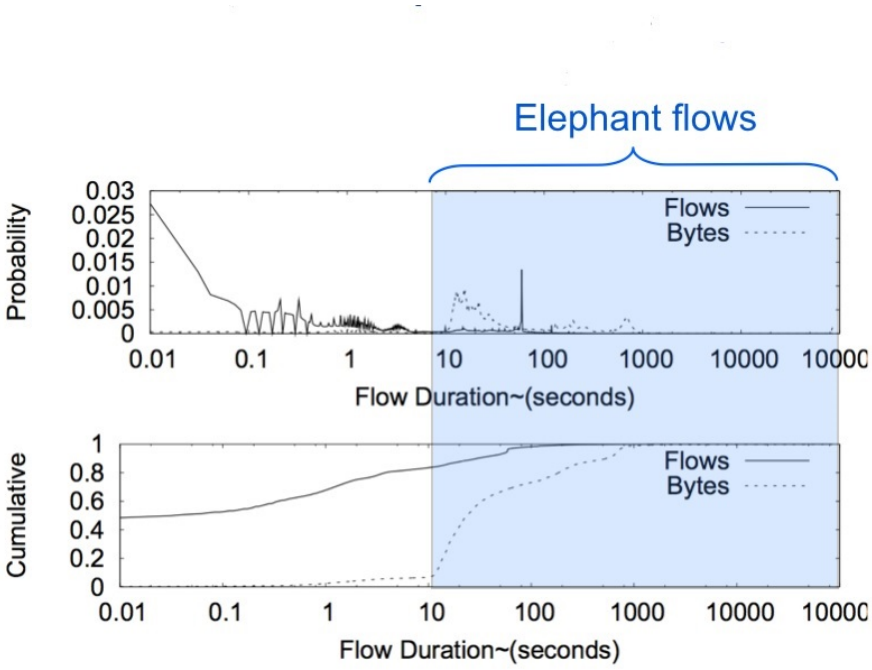

Figure 1.3: Data center traffic measurements published by Microsoft.

The performance impact that elephant flows have on the network can be significant. The temporary load imbalance created by elephant flows can adversely affect any other flow that is trying to utilize the same link common to both flows. This can lead to packet discards which result in an unacknowledged packet at the sender's transport layer being retransmitted when the timeout period expires. The timeout period is usually twice the order of magnitude more than the network's round-trip time, the additional latency can lead to significant variation in performance.

A typical multi-tiered data center network has a significant amount of over subscription, where the hosts attached to the first tier have significantly more provisioned bandwidth between one another than they do with hosts in other tiers. This is necessary to reduce

network cost and improve utilization.

The traffic load between hosts within a cluster of pods is often time critical and bursty in nature. The varying load can create contention and ultimately result in discarded packets for flow control. On the other hand traffic between clusters of pods is typically less time critical, so it can can be staged and scheduled. It is less orchestrated and consists of much larger payloads.

Traffic between data centers is transported over long distance links. The bandwidth costs can be very expensive, therefore it's crucial that when congestion occurs high priority traffic is selected to be transported across these links. Understanding the characteristics of traffic flows is necessary for traffic engineering and capacity planning.

1.3 Data Center Network Architecture

The network topology describes how switches and hosts are interconnected. This is commonly represented as a graph in which vertices represent switches or hosts, and links are the edges that connect them. Representing the network topology as a graph allows for it to be analyzed using sophisticated techniques from graph theory.

The topology of the network is critical in obtaining high throughput. As a data center increases in scale, with large number of network switches connecting thousands of servers, it is very difficult to have all network switches connect to each other in a full mesh. Instead, each switch has a few network ports which it uses to connect to servers or other switches.

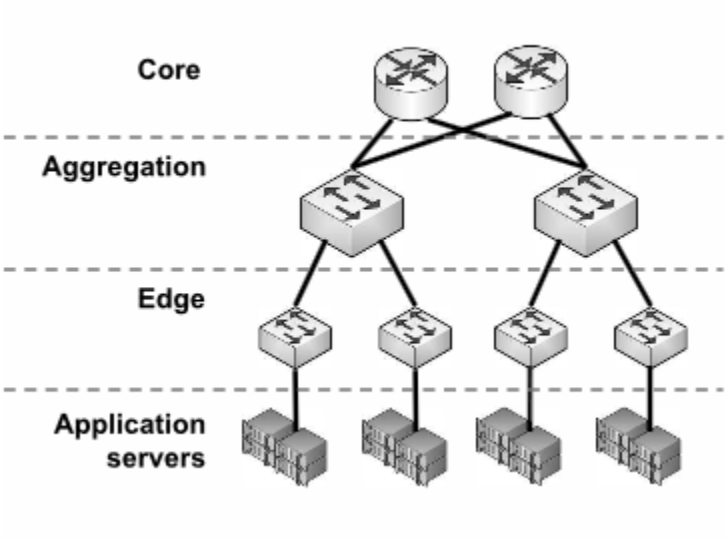

Figure 1.4: Three-tier network model.

A scalable network is one in which increasing the number of ports in the network should linearly increase the delivered bisection bandwidth. Scalability and reliability are inseparable since growing to a large system size requires a robust network.

The amount of traffic the network can carry between servers also depends on the network topology, on how the servers are distributed across the switches, and how the switches are connected to each other.

The design of the network affects a number of design trade offs, including performance, system packaging, path diversity, and redundancy. These trade offs in turn affect the network's resilience to faults and cost.

The typical approach in designing data center networks has been to use the Three-Tier model, shown in Figure 1.4. The Three-Tier model consists of the Edge, Aggregation, and Core

layers. The model is designed to ensure that the network provides a balance of availability, security, flexibility, and manageability. Each tier is focused on specific functions. This allows network operators to choose the right combination of systems and features for each tier.

The Edge tier is used to grant the user, server, or edge device access to the network. The Edge layer generally incorporates switches with ports that provide connectivity to workstations, servers, printers, wireless access points, and so on. It is a common practice to apply features such as security, access control, filters, and quality of service at this tier, which makes it the most feature-rich section of the network.

The Aggregation tier aggregates the wiring closets, using switches to segment workgroups into different networks. It isolates network problems and provides a level of security by acting as a control boundary between the Edge and Core tiers.

The Core tier is used as a high-speed backbone, to switch packets as fast as possible. The Core tier is critical for connectivity; it must provide a high level of availability, scalability and fast convergence.

1.4 Switching and Routing

Switching and routing are the main functions implemented in any network in order to deliver information from one host to another. Switching involves forwarding or filtering frames based on the Media Access Control (MAC) destination address. A network switch will dynamically learn the location of other hosts on the network by logging each learned source

MAC address and the corresponding switch port it was learned on in the switches address table. Switching over Ethernet involves using Address Resolution Protocol (ARP) and Reverse Address Resolution Protocol (RARP) that broadcast messages on the layer 2 network in order to update layer 2 to layer 3 address mappings and vice versa.

Routing involves forwarding and filtering of packets based on the Internet Protocol (IP) destination address. A Router will learn about other networks via routing updates exchanged through routing protocols at layer 3.

A good network topology will have abundant path diversity in which multiple possible paths may exist. Path diversity in the network topology may yield Equal Cost Multipath (ECMP) [18] routing; in that case the routing protocol attempts to load balance the traffic flowing across the links by spreading traffic uniformly. A path through the network is said to be minimal if no shorter path exists. Sometimes a non-minimal path is selected to avoid congestion or to route around a fault.

1.5 Motivation

The rapid evolution of new technologies centered on virtualization has led to the virtualization of servers, storage and most recently networks. The wide adoption of virtualization has been a key enabler of cloud computing.

Cloud computing abstracts the underlying software and hardware recourses, which allows for these resources to be provided and consumed in a more elastic and on demand manner. The

recent proliferation of technologies such as virtualization and cloud computing has led to a tremendous growth of scale for data center networks. It is not surprising to see data centers that support tens of thousands of end-hosts. Resources such as virtual machines and storage volumes are no longer fixed to a certain location, as they can move around the network with very little effort.

The Three-Tier model is no longer suitable for data center networks. The are a number of reasons for this. First, Big Data applications, such as Dryad [45], CIEL [74], MapReduce[31], Hadoop [64], and Spark [96] use computational models that require nodes spread out in different clusters to exchange large amounts of traffic between them. If not carefully engineered, this can lead to contention spots in some parts of the network and light spots in others, therefore current static load balancing algorithms are no longer sufficient for achieving high bandwidth utilization.

Second, the costs for supporting the Tree-Tier model have been on the rise, and show no signs of slowing down [39]; it is therefore necessary to use over-subscription to make efficient use of all available links. Over-subscription factors for links increases as we move from the Edge tier to the Core. This means that different classes of traffic will compete for more available bandwidth as they move towards the Core tier.

Third, with the proliferation of Web 2.0 applications that are mostly built on front-end web servers, a middle-tier application layer, and back-end databases, the direction of traffic has changed from moving "North-South" to more "East-West", which means that these type of applications will have enhanced bandwidth and latency if they exist on the same switch or

layer 2 domain. This results in restrictions on the location of these applications as well as the need for the layer 2 domain to support more traffic.

Forth, the wide use of virtual machines means that switch ports have to learn multiple mac-addresses instead of one, leading to forwarding table exhaustion's in very large environments.

Fifth, introducing servers on the network or relocating them requires careful planning and a number of configuration steps such as, IP addressing and VLAN provisioning, this can take substantial time in large environments.

Sixth, poor support for live seamless mobility, which has become a necessity with virtualization and cloud computing. This means that servers once again have to be on the same layer-2 domain, if they wanted to move around without losing connectivity.

Seventh, frames in layer 2 domains are forwarded along a spanning tree protocol (STP) [32] path in a sub-optimal way, causing some links to be underutilized. This has led to the use of different methods, such as using redundant spanning tree root-bridges with different priorities in an effort to distribute traffic more efficiently, but these methods are neither granular nor flexible.

Eighth, supporting L4-L7 services such as firewalling and load balancing requires careful planning and usually involves steering traffic to dedicated in-line devices. Not only is this time consuming, but also causes choke points at certain locations in the network.

The Three-Tier model lacks a lot of the needed functionality to sustain the growth of the data center and the technologies that are powering the future such as virtualization and

cloud computing.

From an architectural standpoint, traditional data center designs severely limit the full potential of virtualization and cloud computing. Designing an architecture that can meet the needs of today's data center is an active area of research, and as a result a number of architectures have been porpsoed such as, Diverter [47], VL2 [48], VICTOR [50], Portland [53], SecondNet [59], SEC2 [60], NetShare [62], Oktopus [65], CloudNaaS [66], NetLord [73], GateKeeper [77], and SeaWall [78].

1.6 Objective

The broad objective of this work is to propose a new data center architecture, we name OneSwitch, that addresses some of the issues outlined in the previous section.

The specific objectives of the proposed data center architecture can be summarized in the following:

- Simplified provisioning: The ability to plug any device to any point on the network with minimal configuration.
- Equidistant Bandwidth and Latency: Every endpoint has the the same bandwidth and latency available to it to reach any other endpoint on the network in a non-oversubscribed manner.
- High bisection bandwidth between any two nodes on the network.

- Spanning-Tree Free: Spanning tree is eliminated while maintaining redundant paths; hence frames can be routed along the shortest available path, rather than being constrained to a spanning tree.

- Broadcast Free: Broadcasts are eliminated; therefore reducing unwanted traffic loads and allowing greater scalability.

- Dynamic load balancing: The ability to load balance traffic based on traffic volumes and network state.

- Seamless Mobility: The ability to move around form one location to another with minimal interruption.

- Backward Compatibility: The ability to inter-operate with existing hardware and protocols.

1.7 Contribution

The OneSwitch architecture consists of a Folded Clos [1] topology, an OpenFlow central controller, a Location Database, a DHCP Server, and a Routing Service. The OneSwitch components work together to build an Ethernet fabric that appears as a single switch to end devices. This allows the data center to use switches in scale-out topologies to support hosts in a plug and play manner, provide dynamic load balancing, intelligent routing, seamless mobility, equidistant bandwidth and latency.

In this work we have made the following contributions:

- We present a working implementation of OneSwitch that meets the objectives outlined in the previous section.
- We formulate load balancing in OneSwitch as a Supermarket Problem and present a practical version of the problem formulation. We use the practical version of the Supermarket problem to develop a dynamic load balancing algorithm we name Selective Randomized Load balancing (SRL).
- We formulate flow optimization in OneSwitch as a Binary Multicommodity Flow Problem and present Optimized Flow Rerouting (OFR).
- We present a practical flow optimization algorithm we name Flowfit.
- We perform a comparison analysis between ECMP, SRL, OFR, and Flowfit and show that the algorithms we developed achieve better bisection bandwidth and flow completion times.
- We conduct a performance and scalabilty evaluation of OneSwitch and compare it to other proposed architectures.

1.8 Overview

This first chapter described the characteristics of data center networks and their associated applications as well as the challenges with supporting new and emerging application centered around virtualization and cloud computing technologies. In brief, traditional data center architectures lack many features that are required to support the dynamic nature of servers and services. We list the desired objectives and propose the use of OneSwitch, a data center network built using a Folded Clos architecture and a centralized controller, to achieve them.

In chapter 2, we review the background of the material researched in this work. We examine a number of previously proposed approaches and their evolution, as well as introduce concepts and ideas used in this work.

In chapter 3, we present and discuss in detail our proposed OneSwitch architecture.

In chapter 4, we formulate load balancing in OneSwitch as a Supermarket Problem and present SRL.

In chapter 5, we formulate flow optimization in OneSwitch as a Binary Multicommodity Flow Problem and present OFR and FlowFit.

Finally, in chapter 6, we evaluate the performance and scalabilty of OneSwitch and compare it to other proposed methods.

Chapter 2

Background and Related Work

The Ethernet standard has lasted well since its inception with Ethernet frame-structure and addressing remaining ubiquitous. A typical Ethernet network consists of interconnected switches that connect several segments, each with a number of nodes. Each node is identified through a globally unique 48-bit MAC address that is assigned by the manufacture to the network interface card (NIC) of each node.

Each segment is a broadcast medium where NIC's see all packets transmitted on the segment. Ethernet switches deliver a single copy of packets to end-hosts residing on a segment, and prevent unnecessary packet transmissions to other segments; this requires that upon receiving a packet, an Ethernet switch is able to determine at most one port on which to forward the packet.

An Ethernet switch builds a forwarding table where entries consist of a 3-tuple value (MAC

address, Port, Age). Populating the forwarding table is carried out based on the source addresses of packets that a switch receives. Specifically, upon receiving a packet on port P1 with a previously unknown source address SA, the switch creates an entry of (SA, P1, T). This entry is then used for the duration of T to forward frames destined for address SA, out on port P1.

The use of a manufacture-assigned address space allows devices to be plugged anywhere on the network without the need for additional configuration. The downside is now each switch has the task of discovering and storing the location of every addressable device.

The device discovery process in Ethernet switches can cause scalability issues in large environments. There are two main reasons for this. First, the use of a flat address space means that addresses cannot be aggregated. As the number of devices increases so does the size of the forwarding table of each switch. Second, the use of broadcasts as a discovery method for devices on a network is inefficient. As the number of devices increase so does the amount of traffic on the network, which also leads to increased use of CPU cycles by end devices. Also the use of broadcasts in an environment with redundant links can cause broadcast storms, unless switches are running a Spanning Tree Protocol (STP).

In STP environments broadcast packets are transmitted on all ports associated with the links of the logical spanning tree, except for the port on which the packet was received. Although this prevents broadcast storms it does so by blocking redundant links leading to inefficient use of resources.

The traditional method of avoiding the problems with Ethernet has been the artificial subdivision of a network using IP. In an IP network each device is configured with an IP address. Different IP networks are connected to each other using routers. The routers run routing protocols that are used to exchange information about the networks they connect to. This information is used to build forwarding tables on each router. The routers use the destination address of IP packets to do a forwarding table lookup to determine the next hop router.

Unlike Ethernet, IP has a hierarchical address space. Each IP address has a network identifier (network ID) and a host identifier (host ID). The network ID specifies which network a host is on, while the host ID uniquely specifies hosts within a network.

The inherent hierarchy in IP allows blocks of IP addresses to be aggregated. This greatly reduces the size of forwarding tables on routers, allowing IP networks to be more scalable than Ethernet. On the other hand this hierarchy limits the mobility of end devices. This is because the IP address structure contains information about the network where the device is located. When the mobile device travels away from its home location, the system of routing based on IP address "breaks".

In summary, Ethernet networks require no address configuration, allow mobility, but do not scale well. IP networks are more scalable, but harder to configure and limit mobility. The design choices for both Ethernet and IP were made when the networking landscape was much different than it is today.

2.1 Related Work

In this section we first review some of the previously proposed solutions to address the limitation of Ethernet and IP in data center networks and then review some of the solutions that focus on how switches and hosts are interconnected. This is by no means a comprehensive list nor is the intention here to cover all intricate details. The solutions presented here are covered in brief to illustrate the evolution of different approaches. The solutions discussed here also serve as a reference and a motivation to some of the ideas that have been used in our work.

Virtual Private LAN Service (VPLS) [35] is a technology that builds on the strengths of Ethernet and IP. VPLS allows multiple sites to be connected in a single bridged domain over a provider managed IP/MPLS [22] network. All customer sites in a VPLS instance appear to be on the same Local Area Network (LAN).

Rodeheffer et al proposed Smartbridges [20] to solve some of the problems of Ethernet, specifically the issue of sub-optimal link utilization caused by spanning-tree. SmartBridges forward packets between hosts of known location along a shortest possible path in the network and therefore avoid congestion and latency problems that result from traffic being directed towards the root of a spanning tree. SmartBridges operate based on having a complete view of the network topology, which consists of all the interconnection of bridges and segments in the network.

SmartBridges learn the exact segment, to which a host connects to, from neighboring Smart-

Bridges, when a host first transmits a packet. This information is learned through an update protocol that informs all SmartBridges of the location of the host.

In an effort to combine the advantages of both routing and switching, Radia Perlman proposed a method called transparent routing that uses Rbridges [33]. This method is similar to SmartBridges discussed earlier with some implementation differences.

The initial work of Radia Perlman on Rbridges was further enhanced by IETF and submitted as a standard known as Transparent Interconnection of Lots of Links (TRILL) [70]. TRILL uses IS-IS [11] as its links state protocol.

At around the same time the IETF was working on TRILL the IEEE introduced shortest path bridging (SBP) [79], which if formerly known as 801.2aq. SBP is very similar to TRILL with some slight difference in implementation details. Probably the biggest difference is the way frames are encapsulated. TRILL adds a new header while SBP encapsulate the frame in another Ethernet header, which is why it sometimes referred to as MAC-in-MAC encapsulation. This variation in encapsulation, leads to different lookup/forwarding operations for both SPB and TRILL.

Rexford et al proposed SEATLE [42] " A Scalable Ethernet Architecture for Large Enterprises". SEATTLE builds on previous approaches in trying to make Ethernet more efficient. It enables shortest-path forwarding by running a link-state protocol similar to Smartbridges, Rbridges, and TRILL, however, it only distributes switch-level topology information in link-state advertisements, while Host level information is derived through a hashing mechanism.

This host information is maintained in the form of (key, value). Specific examples of these key-value pairs are (MAC address, location), and (IP address, MAC address).

Malcolm Scott et al proposed MOOSE [44] "Multi-level Origin-Organized Scalable Ethernet". MOOSE attributes Ethernet's inability to scale to its flat addressing architecture and therefore focuses on introducing mac address hierarchy in its design. MOOSE switches run a link-state protocol, similar to previously mentioned approaches, to build a switch level view of the topology, this means that each switch will know the address of every other switch in the topology, but what makes MOOSE different is that it uses the switch address to extract the host level information. This is possible because of the way MOOSE addresses are constructed.

Vahdat et al proposed Portland "A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric" [53]. Portland authors make the observation that most data center environments use a multi-rooted network topology that is known and relatively fixed, and hence introduce hierarchical Pseudo MAC (PMAC) addresses as the basis for their design.

Greenberg et al proposed VL2 "Virtual Layer 2" [48]. VL2 uses two separate classes of IP-addresses. The network infrastructure operates using Locator-specific Addresses (LA); all switches and interfaces are assigned LAs, and switches run an IP-based (layer-3) link-state routing protocol that disseminates only these LAs.

The LA's allow switches to obtain complete knowledge about the switch-level topology, as well as forward any packets encapsulated with LAs along the shortest paths. On the

other hand, applications use permanent Application-specific Addresses AAs, which remain unaltered no matter how server's locations change.

Each AA (server) is associated with an LA, the IP address of the top of rack (ToR) switch to which the application server is connected. The VL2 directory system stores the mapping of AAs to LAs, and this mapping is created when application servers are provisioned to a service and assigned AA addresses.

All of the proposals we have reviewed so far focus on addressing the limitations of Ethernet and IP without much attention to the network topology itself. The way servers and switches are connected has a significant impact on scalabilty and performance, hence a number of proposals have explored novel interconnect topologies that can connect massive number of servers and at the same time ensure the agility and robustness of the data center.

Data center networks, based on how servers and switches connect, can be classified as switch-centric, server-centric, or hybrid. A switch-centric network, such as Fat-tree [38], Hypercubes [7], Flattened butterfly [37], Dragonfly [41], and Folded Clos [1] networks, use switches to perform packet forwarding. Server-centric networks, such as CamCube [57], use servers with multiple Network Interface Cards (NICs) to act as switches. Hybrid networks, such as FlatNet [83], DCell [40] and BCube [49], use both switches and servers for packet forwarding. Figures 2.1 and 2.2 show examples of switch-centric and hybrid networks.

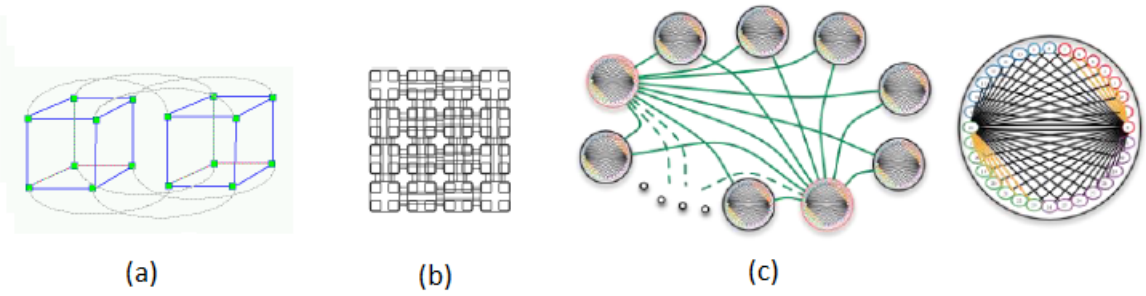

Figure 2.1: Switch-Centric networks (a) Hypercube. (b) Flattened Butterfly. (c) DragonFly.

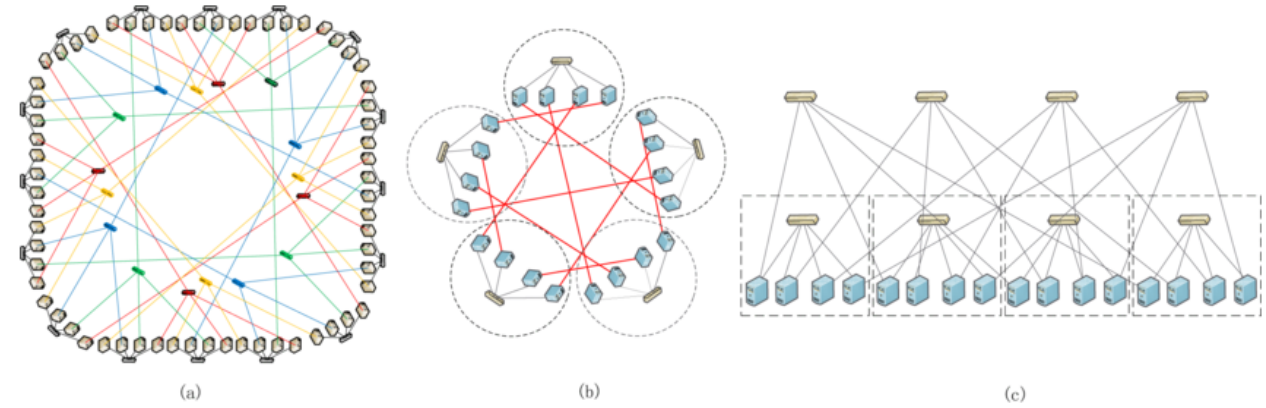

Figure 2.2: Hybrid networks (a) FlatNet. (b) DCell. (c) BCube.

2.2 Three-Stage Clos Switch

Over 70 years ago, telephone switches were constructed from bulky electro-mechanical relays called cross points. In order for these switches to scale, they required a large number of cross points, which increased both cost and complexity. Figure 2.3 shows an N input(output) single stage switch with n^2 cross points.

It wasn't until 1953 when Charles Clos [1] first shed the light on a class of switching networks that were built from smaller switching modules that required fewer cross points, and were

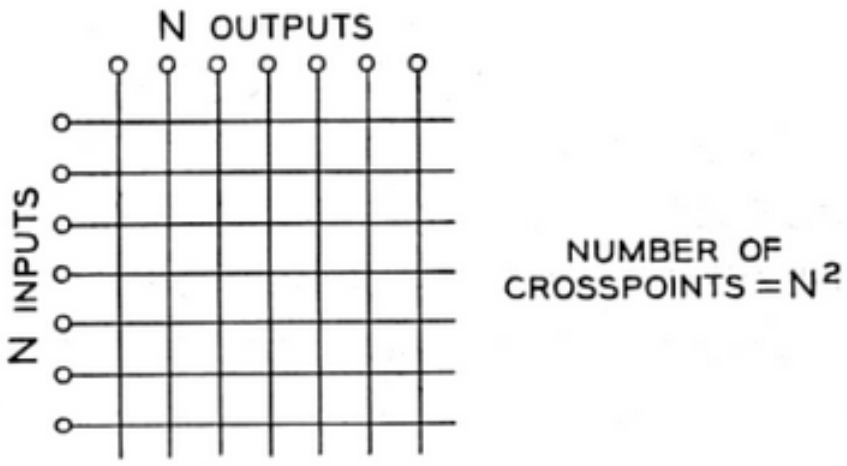

Figure 2.3: Single Stage switch.

immune to blocking. Today, the theory of interconnecting networks based on the Clos topology has been widely used for multiprocessor interconnects and almost all large-scale high performance computing clusters are interconnected with such topologies.

The Three-stage Clos switch, shown in Figure 2.4, consists of three stages: the input stage, the middle stage, and the output stage. There are three key parameters for this network: the number of switch modules in the first and third stages, the number of switch modules in the middle stage and the number of inputs (outputs) to the first (third) stage switch modules. These parameters are commonly denoted by n, m and r respectively and completely characterize the network. We use the notation $C(n, m, r)$ to denote such a network and we let $N = nr$ be the number of network inputs (outputs).

An important performance factor of Three-stage Clos switches is their blocking capabilities. A switch is blocking if there exists a new connection request that cannot be switched. A

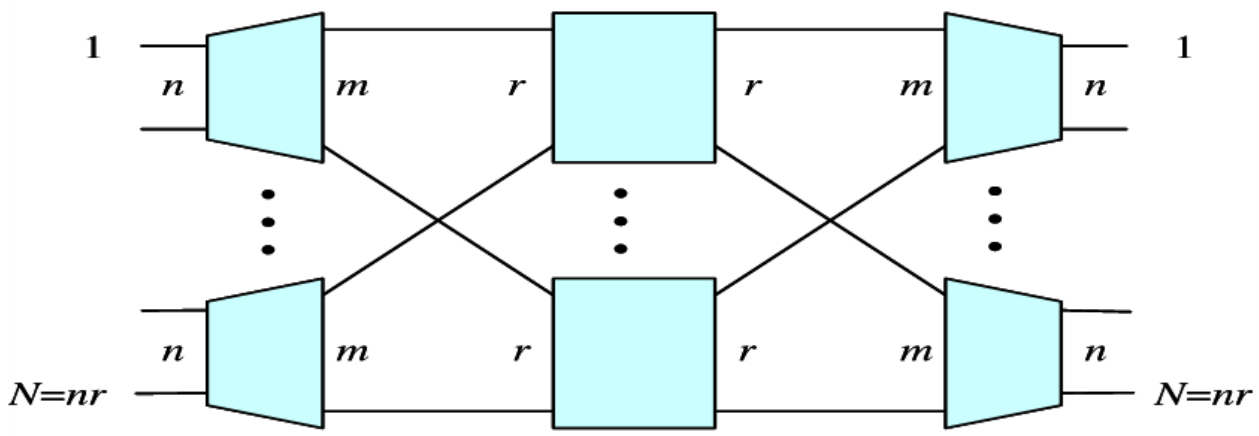

Figure 2.4: Three-Stage Switch Topology.

network is Strictly Non-Blocking (SNB) if it has no blocking state, it is Wide-Sense Non-Blocking (WSNB) if it can avoid getting into a blocking state by leveraging some strategy, and it is Rearrangeably Non-Blocking (RNB) if it can avoid getting into a blocking state by rearranging existing connections.

For circuit-switched systems in which each link can be used by at most one connection at any given time , if $m \geq 2n - 1$ then $C(n, m, r)$ is strictly non-blocking [1]. This allows switching systems to be constructed with less complexity (In terms of cross points).

For example, a $C(n, m, r)$ with $N = 100$, $n = 5$ and $r = 10$ will be Non-blocking if $m \geq 9$. For simplicity if we assume $m = 9$ this means that the first and third stage would require 900 (2x10x5x9) crosspoints, and the middle stage would require 900 (9x10x10) crosspoints, for a total of 1800 crosspoints. This is far less than the number of crosspoints required to to construct a single stage switch, which is $2500(N^2)$ crosspoints.

In packet-switched networks where a link can be used by multiple flows with varying rates, it

has been shown that the number of middle-stage switches required for a Folded Clos switch to be strictly non-blocking is infinite [9].

Although Three-stage Clos switches used in packet switched networks are not SNB they can be WSNB or RNB [9]. In many practical scenarios, the cost of WSNB exceeds the practical benefits they offer. In such situations, one can look to RNB designs for a compromise. Benes [2] showed that in circuit switched networks, Three-stage Clos switches are RNB so long as $m \geq n$. These results were later extended to cover packet switched networks [9]. Since then Three-stage Clos switches have been widely used in packet switched networks because of their nice properties like scalability, path diversity, low diameter, and their simple structure.

A network with a single stage switch has almost ideal network properties: constant latency between all pairs of endpoints as well as full bisection bandwidth (any half of the endpoints can simultaneously communicate with the other half at full line rate). However, Three-stage Clos networks and other multistage switch networks such as, parallel packet switch (PPS) architectures [19], Distributed Switch Architecture (ADSA) [28], and load balanced Birkhoff-von Neuman switches [23] are only able to approximate, but not not truly provide the latency and bisection bandwidth characteristics of single stage networks.

The point-to-point latency in multistage networks is not constant for all port combinations (although it is usually the case that the variance is relatively low). Less obvious, but more important to application performance, is the effect of multistage architecture on network bisection bandwidth, particularly as it is seen by applications.

The bisection bandwidth in multistage networks is defined as the total bandwidth between the two halves of the worst case segmentation's of a network. However, this definition of bisection bandwidth only considers the capacity provided by the hardware. It does not consider how the usage of that capacity may be affected by routing policies. That is, unlike with a single stage switch, traffic must be routed and different communication patterns may require different routing in order to achieve high bisection bandwidth.

There has been a number of studies that have focused on various routing techniques in Three-stage Clos switches by analyzing their performance and convergence time [26, 27].

Three-stage Clos networks outperform many other networks, such as CamCube, BCube, and DCell, when it comes to achieving high bisection bandwidth and low flow completion times, which are both critical for cloud and big data applications. This is because the Three-stage Clos network is an RNB network with very short communication paths. For example in a Three-stage Clos network the maximum path length is 2 and is the same for all endpoints irrespective of which switch they connect to. This allows for endpoints to have the same available bandwidth as well as predictable latency.

In DCell and BCube, as an example, the maximum path length depends on the number of levels k and is $2^{k+1} - 1$ and $k + 1$ for DCell and BCube, respectively. The main advantage of BCube and DCell is the use of recursively defined topologies to scale, but this can also lead to oversubscribed links in topologies with deeper levels.

Due to the nice properties of Three-stage Clos networks there has been a renewed interest

in re-using them in data center networks. From here on we refer to the Three-stage Clos topology as Folded Clos.

2.3 Data Centers with Folded Clos Networks

Figure 2.5 shows an example of a data center network utilizing a Folded Clos topology. The network has two layers of K identical switches with the same port count P. The upper layer is called the core and has $1/3K$ switches. The bottom layer is the edge layer and has $2/3K$ switches. Each edge switch has $P/2$ ports that connect upwards (uplinks) to every core switch, and P/2 ports that connect downwards (downlinks) to servers. The total numbers of servers supported by this topology is therefore $P^2/2$.

The bandwidth ratio between the uplinks and downlinks is an important performance factor. If the uplink bandwidth is greater than downlink bandwidth then the system is undersubscribed, if the uplink bandwidth is less than the downlink bandwidth then the system is oversubscribed and if the uplink bandwidth is equal to downlink bandwidth then the system is balanced.

In this work we focus on balanced systems because of their ability to achieve a high bisection bandwidth. We define bisection bandwidth, in the context of data center networks, as the bandwidth between any two servers whether located on the same switch or on two different switches. The ideal goal of a balanced system is for any two servers to be able to communicate with each other at full capacity, thereby achieving full bisection bandwidth.

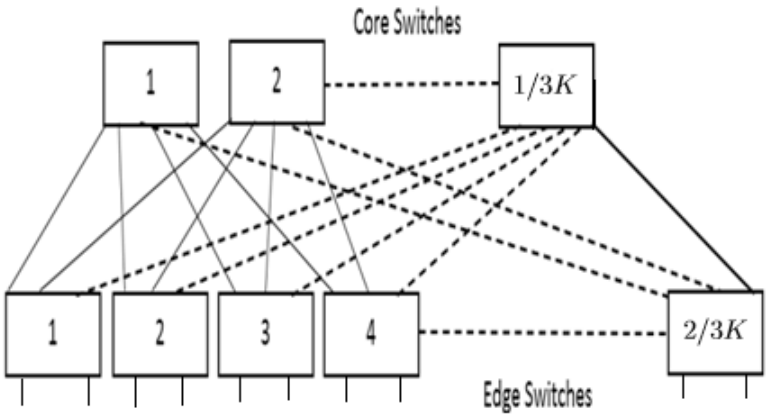

Figure 2.5: Data center network with a Folded Clos topology.

2.4 Load Balancing in Folded Clos Networks

One of the benefits of using Folded Clos topologies is path redundancy. Thus for every source-destination permutation there exist multiple links that can be used. To achieve maximum efficiency, reduce congestion, and minimize packet loss, load balancing is necessary among links. The question now becomes how we actually distribute packets across these multiple links. If packets are distributed evenly we can better utilize link capacities; however, distributing packets over multiple links naturally causes out-of-order delivery, which can degrade performance especially for TCP applications [46].

Static hashing offers a simple solution for maintaining packet order by sending a flow over a unique link. A hash is applied to flow headers, and the result is used to determine link assignment. Most data centers today rely on static hashing such as layer-3 ECMP for their load balancing. Another similar method is Valiant Load Balancing (VLB) [6]: The

scheme works by routing through a randomly picked intermediate node en route to a packet's destination.

Static hashing and VLB are forms of randomized load balancing that are relatively simple to implement and have worked well in the past when traffic patterns weren't so diverse. Today, because of large and many persistent flows, these methods are not as effective, due to multiple flows ending up on one link and causing congestion. The congestion will not be resolved until one or more flows are completed.

We use a simple example to clarify the the impact of randomized load balancing on large flows. Figure 2.6 shows a small 2x4 Folded-Clos network with two core switches and four edge switches. The uplinks and downlinks all have speeds of 1 Gbps. Server A has 600 Mbps and 400 Mbps flows to send to server C. Server B has 800 Mbps and 200 Mbps flows to send to Server D. We can achieve full bisection bandwidth, as long as we ensure that the two large flows, F1 (600 Mbps) and F3 (800 Mbps), do not end up on the same uplink. If they do, their combined bandwidth would be 1.4 Gbps, which is greater than the 1 Gbps capacity of the uplink. This example illustrates the importance of flow placement in achieving high bisection bandwidth.

Dynamic hashing is another form of load balancing that uses a hash value, but instead of using the hash to assign the flows to links, it first determines an intermediate bin number and assigns flows to bins. The flows are dispatched from the bins according to a defined policy and then are assigned to links. For example, in Open Shortest Path Optimized Multipath protocol (OSPF-OMP) [15], the bins are assigned proportionally to the links bandwidth.

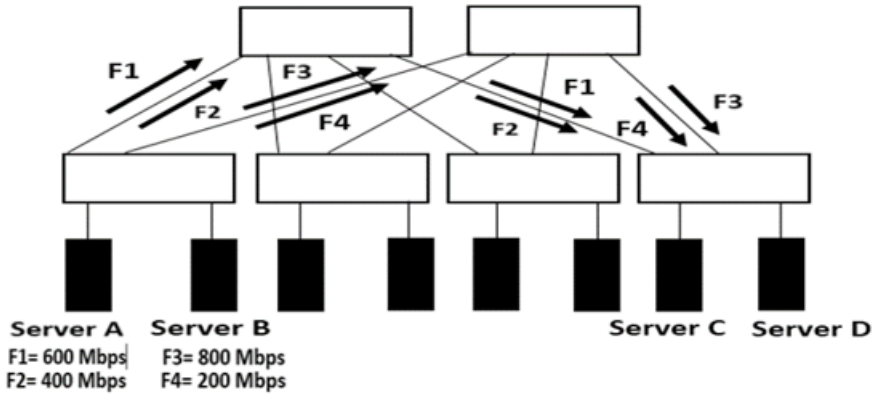

Figure 2.6: 2x4 Folded Clos Network.

Although load balancing and blocking in Folded Clos architectures have been extensively studied [9, 14, 10, 25], there is a renewed interest in studying load balancing in Folded Clos topologies in the context of data center networks.

2.5 ECMP and Related Work

Currently most data center networks deploy routing protocols that rely on ECMP to evenly spread traffic across redundant paths. ECMP performance relies on flow granularity and it can be effective and achieve near-optimal bisection bandwidth when flows are many and short lived (Mice Flows), but not so effective when flows are large and long lived (Elephant flows). This is because when a hash-collision occurs in ECMP, two or more elephant flows end up on the same link, creating unavoidable congestion.

The poor performance of ECMP can be attributed to two key issues:

1. Hash collision of elephant flows.

2. No knowledge of network state.

There has been a number of proposals to address these two issues. For the first issue, Presto [98] divides flows into equal-size "flowcells" at the hosts. The hosts then proactively route the cells via source routing. In [98] they argue that in a symmetric network topology where all flows are mice, ECMP provides near optimal load balancing. They also deploy a centralized controller to react to occasional asymmetries in topology such as failures.

Other efforts include dividing flows into "flowlets" [36] and balancing flowlets instead of flows, or per-packet spreading of traffic in a round robin fashion [85, 87]. Presto's choice of flowcells (64KB pieces of flows) is motivated by the fact that flowlets are coarse grained, and is dictated by the practical challenges of performing per-packet load balancing in the hosts.

The common approach among the work addressing hash collisions is dividing large flows into small chunks and routing them separately in a proactive manner, with no need for information about the network state.

For the second issue, Planck [93] presents a fast network measurement architecture that enables rerouting congested flows in milliseconds. Fastpass [92] positions that each sender should delegate control to a centralized arbiter to dictate when and through which path each packet should be transmitted. Hedera [58], MicroTE [67], and Mahout [68] re-route elephant flows to compensate for the inefficiency of ECMP hashing them onto the same path.

Other methods take a hybrid approach by both splitting traffic into flows and using informa-

tion about the network to allocate flows to paths. Some approaches use global information gathered from all of the switches, such as CONGA [91], while others rely on just local information such as Drill [97].

2.6 Mininet

Mininet [99] is an open source network emulator that uses Linux container based virtualization to emulate hosts, links, switches, and applications, using a real Linux kernel [81], in software rather than hardware.

Mininet uses lightweight virtualization to make a single system look like a complete network running, the same kernel, system, and user code. Because Mininet uses a single Linux kernel for all virtual hosts, software that depends on BSD, Windows, or other operating system kernels are not supported. By default all Mininet hosts share the host file system and PID space.

A Mininet host behaves just like a real machine. It can run real Linux programs, such as iperf [110], that send packets through what seems like a real Ethernet interface, with a given link speed and delay. Packets get processed by what looks like a real Ethernet switch with a given amount of queueing. The packets can even be monitored with tools like Wireshark.

Mininet commands can be used to quickly create custom topologies: a single switch, a data center, or larger Internet-like topologies. Topologies can also be created using graphical tools such as the the Mininet Editor shown in Figure 2.7.

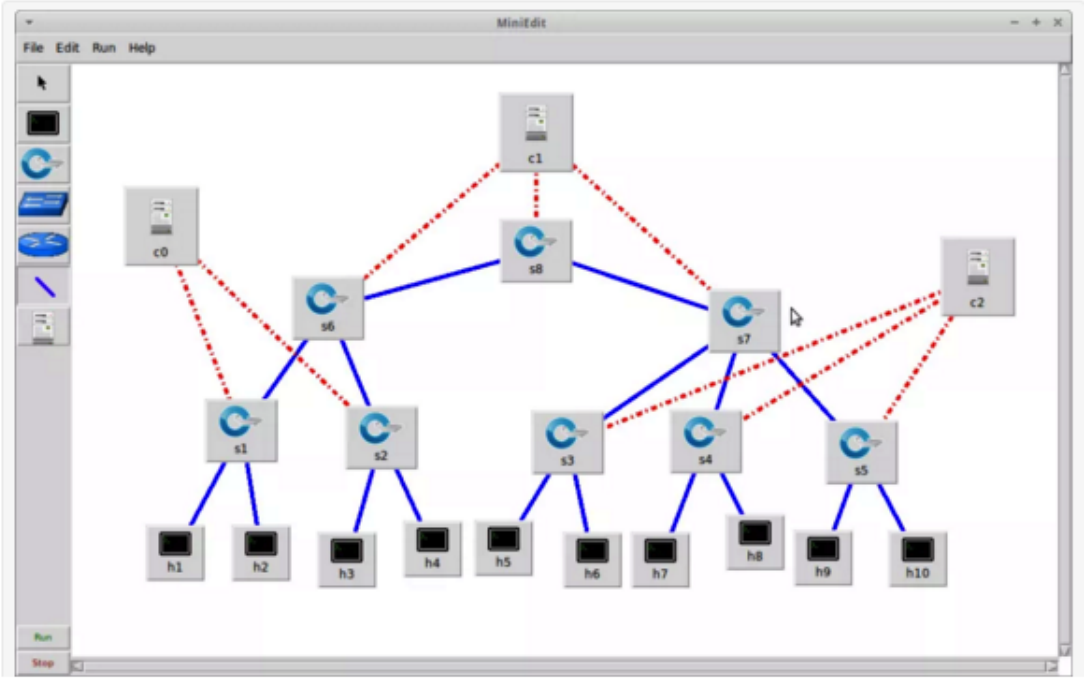

Figure 2.7: Mininet Editor.

A Mininet network consists of the following components:

- Isolated Hosts: An emulated host in Mininet is a group of user-level processes moved into a network namespace. Network namespaces provide process groups with exclusive ownership of interfaces, ports, and routing tables (such as ARP and IP). For example, two web servers in two network namespaces can coexist on one system, both listening to private eth0 interfaces on port 80. Mininet uses CPU Bandwidth Limiting to limit the fraction of a CPU available to each process group.

- Emulated Links: The data rate of each link is enforced by Linux Traffic Control (tc), which has a number of packet schedulers to shape traffic to a configured rate. Each

emulated host has its own virtual Ethernet interface(s) (created and installed with ip link add/set). A virtual Ethernet (or veth) pair, acts like a wire connecting two virtual interfaces, or virtual switch ports; packets sent through one interface are delivered to the other, and each interface appears as a fully functional Ethernet port to all system and application software.

- Emulated Switches: Mininet typically uses the default Linux bridge or Open vSwitch (OVS) [100] running in kernel mode to switch packets across interfaces. Switches can run in the kernel if high performance is required or in user space if the switches need to be easily modified.

Chapter 3

OneSwitch

A typical communications network consists of a variety of network devices such as, routers, switches, firewalls, load balancers,...etc. Each device implements it's own proprietary control stack and provides a vendor-dependent management interface in the form of a Command Line Interface (CLI), a web interface, or a management protocol. Some specific controls, such as STP [32] and SPB [79] are handled by other complex protocols.

Each additional component that is added to a device's control stack increases the complexity and complicates integrated network management. The consequences are often low network utilization, poor manageability, lack of control options in cross-network configurations, and vendor lock-in.

A network concept that has come to recent prominence is Software-Defined Networking (SDN) [72]. SDN separates the data forwarding and control functions of networking devices,

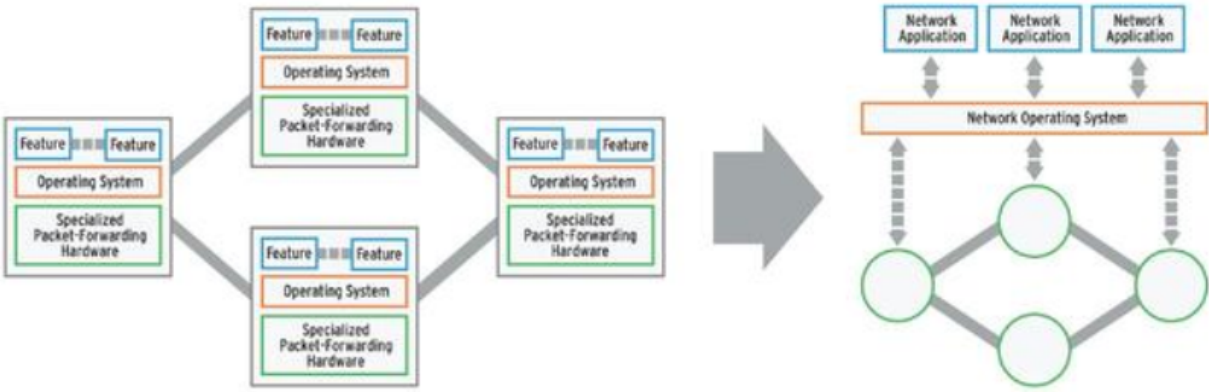

Figure 3.1: Legacy network vs SDN.

such as routers and switches, with a well-defined Application Programming Interface (API) between the two.

SDN converts legacy networks with distributed, heterogeneous, and vertically integrated components into networks with centralized, uniform, and direct control of the infrastructure, thus removing the need for complex and sophisticated network management. SDN also provides added flexibility and freedom from the proprietary protocols of a single hardware vendor. Figure 3.1 shows the difference between a legacy network and SDN.

The SDN framework, shown in Figure 3.2, consists of 3 layers, Application, Control, and Infrastructure. The Application layer enables network administrators to respond quickly to changing business requirements via an SDN controller implemented in the Control layer. The SDN controller is the "brains" of the network. It provides a centralized view of the overall network, and enables network administrators to dictate to the underlying systems how the forwarding plane should handle network traffic.

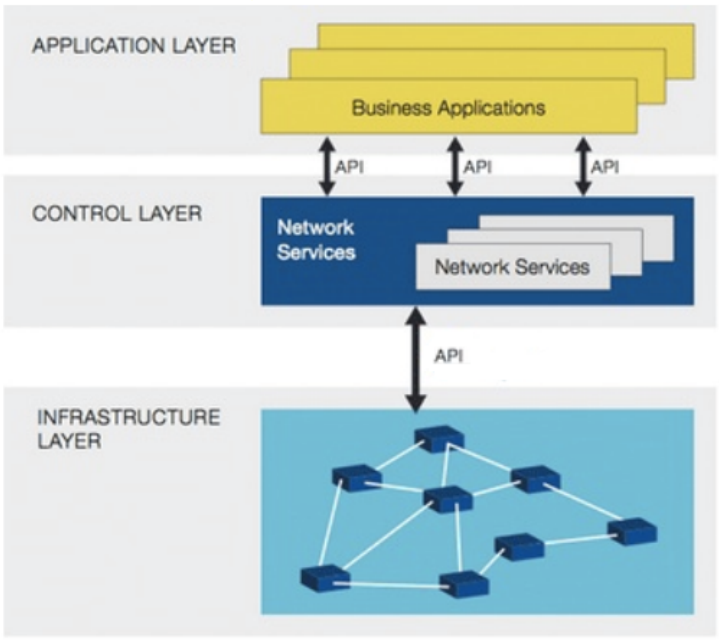

Figure 3.2: Software-Defined Networking (SDN) framework.

The Control layer uses southbound and northbound APIs to communicate with the Application and Infrastructure layers, respectively.

3.1 OneSwitch Building Blocks

3.1.1 OpenFlow

To implement SDN, two requirements must be met. First, there must be a common logical architecture in all switches, routers, and other network devices to be managed by an SDN controller. This logical architecture may be implemented in different ways on different vendor equipment and in different types of network devices, so long as the SDN controller sees a

uniform logical switch function. Second, a standard, secure protocol is needed between the SDN controller and the network device.

Both of these requirements are addressed by OpenFlow [82], which is both a protocol between SDN controllers and network devices, as well as a specification of the logical structure of the network switch functions [80].

OpenFlow provides an open protocol to program the flow table in different switches and routers. An OpenFlow switch, shown in Figure 3.3, consists of one or more flow tables and a group table, which perform packet lookups and forwarding, and an OpenFlow channel to an external controller. The controller manages the switch via the OpenFlow protocol. By Using the OpenFlow protocol, the controller can add, update, and delete flow entries, both reactively (in response to packets) and proactively.

Each flow table in the switch contains a set of flow entries; each flow entry consists of match fields, counters, and a set of instructions to apply to matching packets.

Matching starts at the first flow table and may continue to additional flow tables. Flow entries match packets in priority order, with the first matching entry in each table being used. If a matching entry is found, the instructions associated with the specific flow entry are executed. If no match is found in a flow table, the outcome depends on switch configuration: the packet may be forwarded to the controller over the OpenFlow channel, dropped, or may continue to the next flow table.

The OpenFlow protocol supports three message types, controller-to-switch, asynchronous,

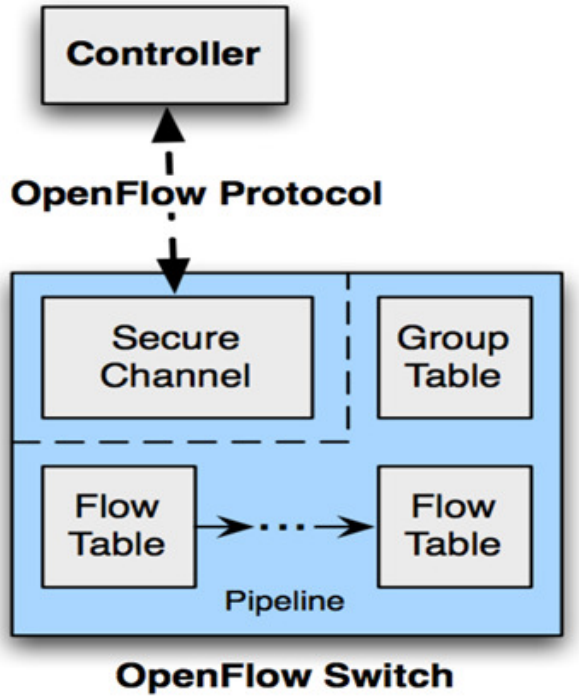

Figure 3.3: OpenFlow Switch Architecture.

and symmetric, each with multiple sub-types. Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch. Asynchronous messages are initiated by the switch and used to update the controller of network events and changes to the switch state. Symmetric messages are initiated by either the switch or the controller and sent without solicitation.

OpenFlow maintains counters for each flow table, flow entry, port, or queue. Figure 3.4 shows the set of counters defined by the OpenFlow specification. A switch is not required to support all counters, just those marked "Required". The Duration refers to the amount of time the flow entry, a port, a group, or a queue has been installed in the switch, and is tracked in seconds precision.

Counter	**Bits**	
Per Flow Table		
Reference Count (active entries)	32	*Required*
Packet Lookups	64	*Optional*
Packet Matches	64	*Optional*
Per Flow Entry		
Received Packets	64	*Optional*
Received Bytes	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Port		
Received Packets	64	*Required*
Transmitted Packets	64	*Required*
Received Bytes	64	*Optional*
Transmitted Bytes	64	*Optional*
Receive Drops	64	*Optional*
Transmit Drops	64	*Optional*
Receive Errors	64	*Optional*
Transmit Errors	64	*Optional*
Receive Frame Alignment Errors	64	*Optional*
Receive Overrun Errors	64	*Optional*
Receive CRC Errors	64	*Optional*
Collisions	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*
Per Queue		
Transmit Packets	64	*Required*
Transmit Bytes	64	*Optional*
Transmit Overrun Errors	64	*Optional*
Duration (seconds)	32	*Required*
Duration (nanoseconds)	32	*Optional*

Figure 3.4: OpenFlow Counters.

The presence of many statistics allows an OpenFlow controller to query the switch for information about its running state. The information collected represents real-time statistics at the time the switch was queried and may not be the same by the time the controller receives the information. This is because of possible delay between when information was queried and when it was received. Fortunately, for many applications, this slight inaccuracy is tolerable or negligible, especially if the controller is on it's own control network and is located in close proximity to switches.

OpenFlow statistics are mainly byte counters. These raw statistics have no notion of time. To determine some statistic that is dependent on time, such as bandwidth, the controller can use byte counters returned at two points in time. The difference between these two counters divided by the time elapsed between the two counter values yields the bandwidth. The accuracy of the approach is dependent on the time stamping of statistics collection, control plane latency, and the statistics collection interval.

The time stamp of when the statistics were collected is not included in the statistics reply sent by the switch. As such, the controller must note when the statistics request was sent and when the statistics reply arrived. The control plane latency will of course impact the time stamp recorded as well.

An OpenFlow switch must be able to establish communication with a controller at a user-configurable IP address, using a user-specified port. If the switch knows the IP address of the controller, the switch initiates a standard TLS or TCP connection to the controller.

The availability of OpenFlow devices [43, 106, 109, 112] has allowed for OpenFlow to be adopted in many data center designs [58, 61, 53]. In addition, the availability of virtual switches makes it practical to experiment with and deploy new algorithms.

3.1.2 SDN Controller

An SDN controller is the application that acts as a strategic control point in the SDN network. It manages flows in switches/routers via southbound APIs and the applications

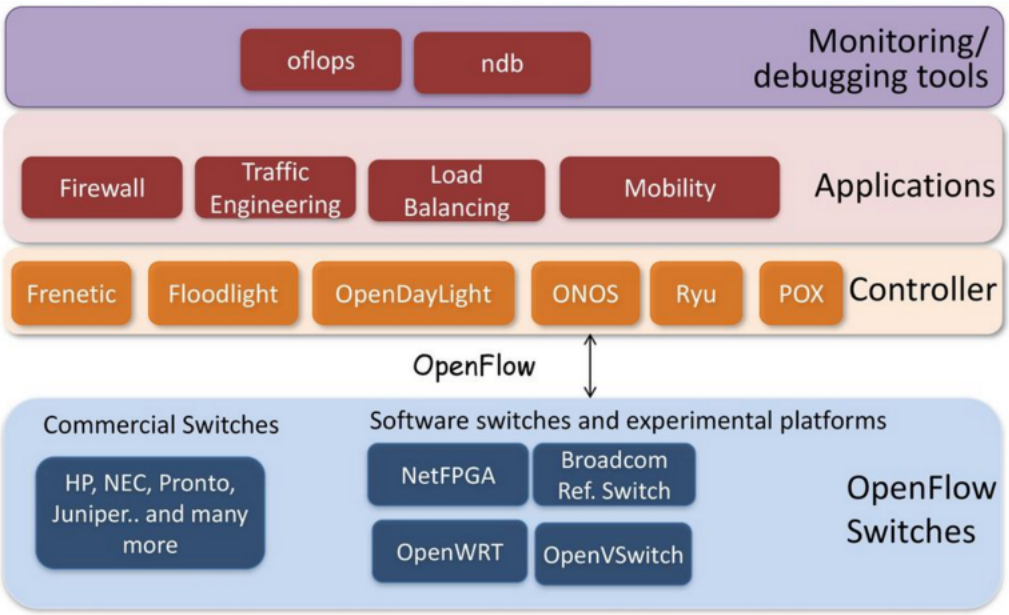

Figure 3.5: OpenFlow Controllers.

and business logic via northbound APIs. An SDN controller platform typically contains a collection of "pluggable" modules that can perform different network tasks. Some of the basic tasks including inventorying what devices are within the network and the capabilities of each, gathering network statistics, etc. Extensions can be inserted that enhance the functionality and support more advanced capabilities, such as running algorithms to perform analytics and orchestrating new rules throughout the network.

OpenFlow controllers, shown in Figure 3.5, are a type of SDN controllers that use the OpenFlow protocol. The following is a list of the most commonly used open source controllers.

- OpenDaylight [116]: A Java-based multi-protocol SDN controller that's based on OSGi [118], as well as an OpenFlow plug-in, an OpenFlow protocol library, the OVS Database Management Protocol [90] and YANG [55] tools.

- Floodlight [107]: A Java-based openflow controller that was forked from the Beacon [89] controller and is Apache-licensed. The controller can handle mixed OpenFlow and non-OpenFlow networks. The controller includes support for the OpenStack [117] cloud orchestration platform as well.

- Ryu [122]: An SDN controller that supports various protocols for managing network devices, including OpenFlow, Netconf [71] and OF-config [114].

- ONOS [115]: An SDN controller that is focused on service provider use-cases. The platform is written in Java and uses OSGi for functionality management.

- POX [101]: A Pyhton-based SDN controller developed by Stanford university to replace the original NOX [113] controller that was written in Java.

In our implementation of OneSwitch we use the POX controller. Although Any of the above OpenFlow controllers could have been used, we found that the POX controller is the most suitable. Next, we discuss the reasons for choosing the POX controller as well explain some of its inner workings.

POX Controller

POX is the newer, Python-based version of NOX (or NOX in Python). POX is often used in academic network research to develop SDN applications such as network protocol research. One side effect of its widespread academic use is that example code is available for a number networking applications. The code for these applications can be used as starter code for

various other projects and experiments. For example, POX has reusable sample components for path selection, topology discovery, and so on.

POX runs on Linux, Mac OS, and Windows and can be bundled with install-free PyPy [121] runtime for easy deployment. It also supports a number of GUI and visualization tools. POX supports OpenFlow 1.0 and 1.1 switches and includes special support for OVS.

The POX controller has a number of software components that can be used to perform SDN functions. More complex SDN functions can be performed by creating new software components. These software components can be invoked when POX is started from the command line. POX also contains a number of APIs to help develop network control applications.

Switches communicate with the POX controller though OpenFlow messages defined in the OpenFlow Specification. These messages show up in POX as events. There's an event type corresponding to each message type that a switch might send. Event handlers can be written to deal with each type of event.

POX has a library for parsing and constructing packets. The library has support for a number of different packet types. Application can be built to construct packets and send them out of a switch, or receive them from a switch.

The POX controller can be configured with rules. When a packet sent to the controller matches a rule, an OpenFlow action is applied to packets that match the rule.

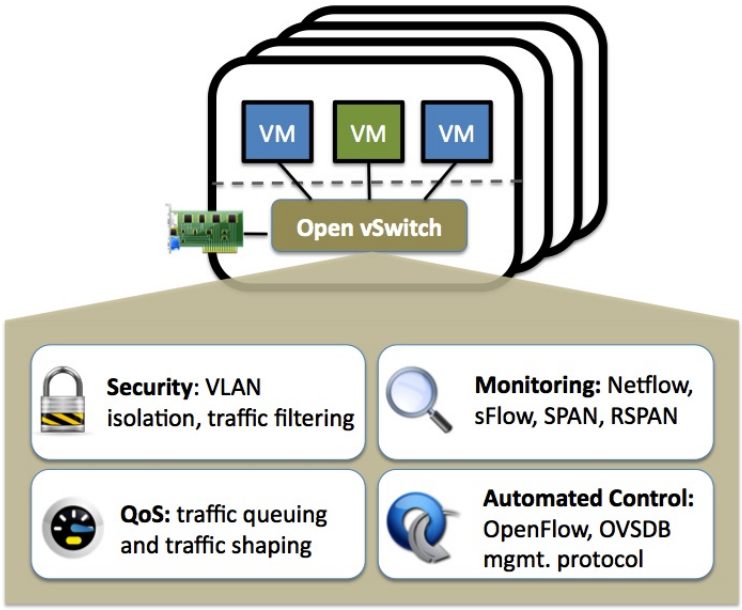

Figure 3.6: Open vSwitch.

3.1.3 Open vSwitch

Traditionally servers physically connected to a hardware-based switch located in the data center. With the proliferation of virtualization that is no longer the case. It's common now to have virtual machines (VM) connecting to a virtual switch. The virtual switch is a software layer that resides in a server that is hosting the VM. VMs have logical or virtual Ethernet ports. These logical ports connect to the virtual switch.

Open vSwitch (OVS) [100] is an multi layer virtual switch created by the open source community and licensed under Apache 2.0. OVS is included in the Linux 3.3 Kernel by default and most users space utilities are available in Linux distributions. From a control and management perspective, OVS leverages OpenFlow and the Open vSwitch Database (OVSDB)

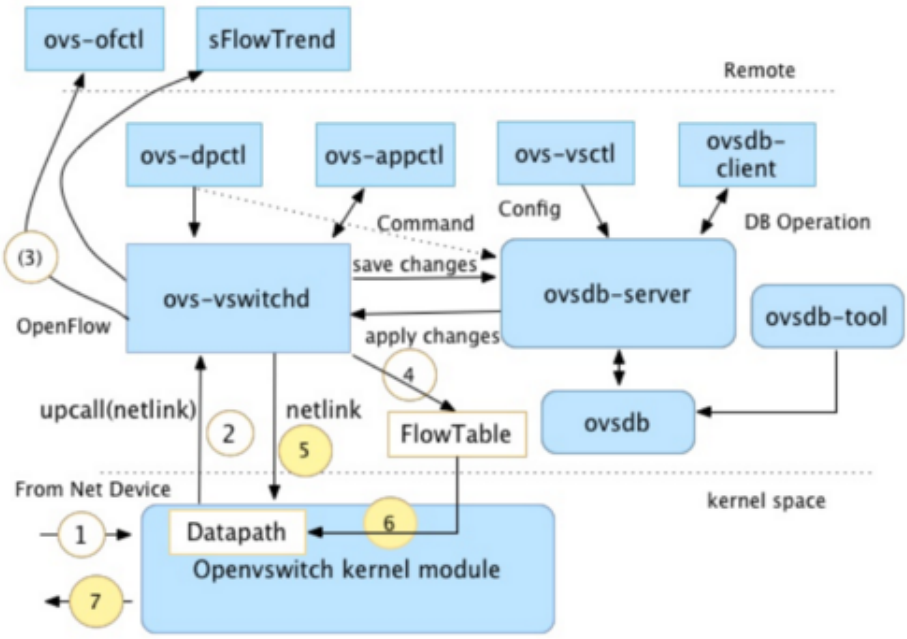

Figure 3.7: Open vSwitch internal architecture.

management protocol, which means it can operate both as a soft switch running within the hypervisor, and as the control stack for switching silicon. OVS allows for network automation through programmatic extensions and supports many features such as NetFlow, sFlow, LACP, QoS....etc.

The OVS distribution, shown in Figure 3.7, consists of the the following components:

- ovs-ofctl: A utility for querying and controlling OpenFlow switches and controllers.
- sFlowTrend: A tool that generate real-time displays of the top users and applications making use of network bandwidth.
- ovs-dpctl, a tool for configuring the switch kernel module.

- ovs-appctl: A utility that sends commands to running Open vSwitch daemons.
- ovs-vsctl: A utility for querying and updating the configuration of ovs-vswitchd.
- ovsdb-client: A utility for using JSON-RPC client calls to the ovs-vswitchd.
- ovs-vswitchd: A daemon that implements the switch, along with a companion Linux kernel module for flow-based switching.
- ovsdb-server: A lightweight database server that ovs-vswitchd queries to obtain its configuration.
- ovsdb-tool: A command line tool to manage ovsdb-server.
- ovsdb: A tool that persists the data across reboots.
- Openvswitch kernel module: Designed to be fast and simple. Handles switching and tunneling in the datapath.

Open vSwitch can be used as an SDN switch by using OpenFlow to control data forwarding. OpenFlow allows an SDN controller to add, remove, update, monitor, and obtain statistics on flow tables and their flows, as well as to divert selected packets to the controller and to inject packets from the controller into the switch.

In Open vSwitch, ovs-vswitchd receives OpenFlow flow tables from an SDN controller, matches any packets received from the datapath module against these OpenFlow tables, gather the actions applied, and finally caches the result in the kernel datapath. This allows

the datapath module to remain unaware of the particulars of the OpenFlow wire protocol, further simplifying it. From the OpenFlow controller's point of view, the caching and separation into user and kernel components are invisible implementation details: in the controller's view, each packet visits a series of OpenFlow flow tables and the switch finds the highest priority flow whose conditions are satisfied by the packet, and executes its OpenFlow actions.

3.2 OneSwitch Architecture

The OneSwitch architecture, shown in Figure 3.8, is built using the SDN framework. The OneSwitch architecture consists of a control plane and a data plane. The control plane is responsible for making all of the forwarding decisions while the data plane does the actual forwarding of packets. In conventional networking, both planes are implemented in the firmware of network switches. In OneSwitch the control and data plane are decoupled.

The OneSwitch control plane consists of a POX Controller, DHCP Server, Routing Service, and a Location Database. The OneSwitch data plane consists of a number of OVSs connected through a Folded Clos topology.

The OneSwitch architecture components work together to virtualize the underlying Folded Clos network and create the illusion that hosts are connected to one data center switch, hence the name OneSwitch. The components of OneSwitch provide all of the services needed such as address assignment, packet forwarding, routing, load balancing, mobility, and congestion management.

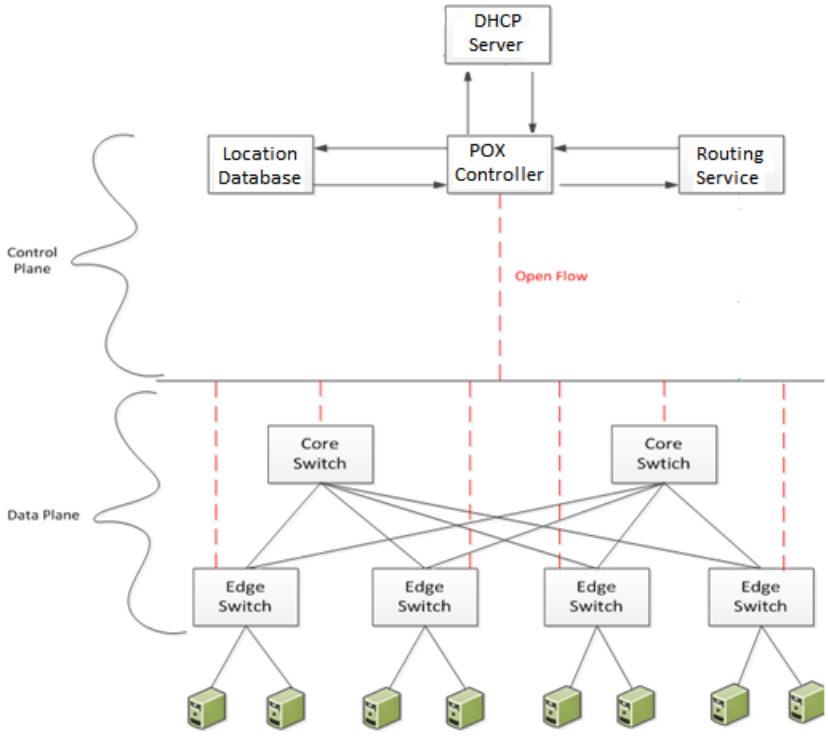

Figure 3.8: OneSwitch Architecture.

When an OVS first comes online it establishes a TCP connection to the POX controller. The POX controller sends a feature request message to the OVS and waits for a reply. When the reply reaches the POX controller the controller learns about the features provided by the OVS. One important feature is the Datapath Identifier (DPID). The DPID is defined as a 64 bit number in the OpenFlow specification. The DPID uniquely identifies a data path. The lower 48 bits are intended for the switch MAC address, while the top 16 bits can be configured by the operator.

The POX controller periodically instructs the OVSs to flood Link Layer Discovery Protocol (LLDP) [51] and Broadcast Domain Discovery Protocol (BDDP) [107] packets through all of

their ports. The discovery protocol packet contains the DPID of the sender along with the port of the switch that the message originates from. The reserved set of destination MAC addresses and EtherTypes used by the discovery protocol packets allows the POX controller to differentiate them from other data packets. LLDP is used to discover direct links between switches and BDDP is used to discover the switches in the same broadcast domain.

By using a combination of LLDP and BDDP packets, the POX controller discovers the direct and indirect connections between the switches. Once the POX controllers learns the topology, the shortest path is found using the Routing Service.

The Routing Service is used to route packets between switches (DPID's). It uses a Bidirectional Depth First Search (BDFS) [8] algorithm to find the shortest route between the source DPID and the destination DPID.

In our implementation of BDFS the main data structures used are path dictionaries, one for each the source and destination. Each path dictionary has node ID's as its keys and a list of routes as it's value, where each route records the list of DPID's to get from the starting point (source or destination) to the key. BFDS is used to select the shortest path, but since all destinations have the same cost and are two switches away, BDFS will always return a list of all available paths between source and destination DPID's. A path is then selected from the list based on load balancing algorithm used.

The Routing Service has 2 load balancing algorithms that can be used to select paths.

1. ECMP

2. SRL

The Routing Service also uses 2 flow optimization algorithms that can be used to re-route flows.

1. OFR

2. FlowFit

A detail description of SRL is presented in sections 4.5 whereas OFR and FlowFit are presented in sections 5.4.1 and 2, respectively.

The Location Database contains host MAC address and IP address to DPID mappings. The mappings bind hosts to OVSs. These binding are populated by the POX controller through DHCP and ARP requests. This guarantees that if devices move their mappings are updated.

The POX controller acts as a DHCP relay agent, forwarding DHCP requests to the DHCP server. We use the proto.dhcpd module [119] to implement the DHCP server. The DHCP server is configured to assign IP addresses that belong to the same IP Subnet.

The POX controller also acts as an ARP proxy. When a host sends a packet for the first time, the networking stack on the host generates a broadcast ARP request for the destination. The ARP request is intercepted by the edge switch and sent to the POX controller. The POX controller responds back with an ARP reply once the destination mac address is resolved. Because most end hosts send GARP or RARP requests to announce themselves to the network right after they come online, the POX controller will immediately have the

opportunity to learn their MAC and IP addresses and distribute this information to other Edge switches. As a result, ARP flooding caused by host ARP learning behavior is reduced.

The OVSs forward packets received from hosts according to OpenFlow rules that have been programmed by the POX controller. If no matching rules are found, the packets are forwarded to the POX controller to determine how packet forwarding will be handled.

3.3 OneSwitch Packet Forwarding

Next we explain how packets are forwarded between hosts when connected to OneSwitch. We discuss 3 scenarios that cover all of the possible communications between hosts. We use a simple OneSwitch topology with 2 core switches and 4 edge switches to illustrate how different components of OneSwitch interact in order to perform packet forwarding.

3.3.1 Packet forwarding between hosts on the same edge switch

Figure 3.9 shows the steps for packet forwarding between Host A and Host B. Both hosts are connected to edge switch 1. The steps are as follows:

1. When Host A(or B) first come online it sends a DHCP request.

2. Edge switch 1 intercepts the DHCP request and sends it to the POX controller.

3. The POX controller records the DPID of edge Switch 1 and the source Mac address

(Mac A or Mac B) of the DHCP request and forwards it to the DHCP server.

4. The DHCP server assigns an IP address and sends the response back to the POX controller.

5. The POX controller records the assigned IP address (IP A or IP B) and forwards the response back to edge switch 1.

6. Edge switch 1 forwards the DHCP reply back to the host that initiated the request.

7. Through the DHCP requests the POX controller learns Host A and B, Mac addresses, IP addresses, and the DPID address of edge switch 1. The POX controller stores these bindings in the Location Database.

8. When Host A communicates with Host B for the first time, Host A will send an ARP request.

9. Edge switch 1 intercepts the ARP request and sends it to the POX controller.

10. The POX controller will query the Location Database.

11. The Location Database will return the binding for Host B to the POX controller.

12. The POX controller will use the Mac address in the Host B binding to construct an ARP reply.

13. Edge switch 1 forwards the reply to Host A.

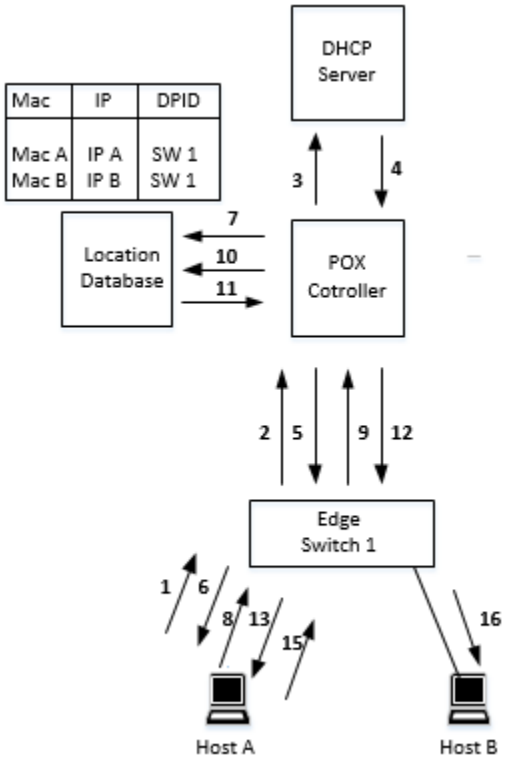

Figure 3.9: Packet forwarding between hosts on the same edge switch.

14. The POX controller checks the DPID address in the Host B binding and determines that Host B is located on edge switch 1. Because Host B is on the same edge switch the Routing service is not invoked.

15. Host A forwards packets destined to Host B to edge switch 1.

16. Edge switch 1 consults its local MAC address table to determine which port Host B is connected to and forwards the packets to Host B.

3.3.2 Packet forwarding between hosts on different edge switches

Figure 3.10 shows the steps for packet forwarding between Host A and Host B. Hosts A is connected to edge switch 1, while Host B is connected to edge switch 2. The steps are as follows:

1. When Host A and B first come online they send a DHCP request
2. Edge switch 1 and 2 intercept the DHCP request and sends it to the POX controller.
3. The POX controller records the DPID of edge switch 1, 2, and the source Mac address of the DHCP request and forwards it to the DHCP server.
4. The DHCP server assigns an IP address and sends the response back to the POX controller.
5. The POX controller records the IP address and forwards the response back to edge switch 1 and 2.
6. Edge switch 1 and 2 forward the DHCP reply back to the host that initiated the request.
7. Through the DHCP requests the POX controller learns the Host Mac addresses, IP addresses, and DPID addresses of edge switch 1 and 2. The POX controller stores these bindings in the Location Database.

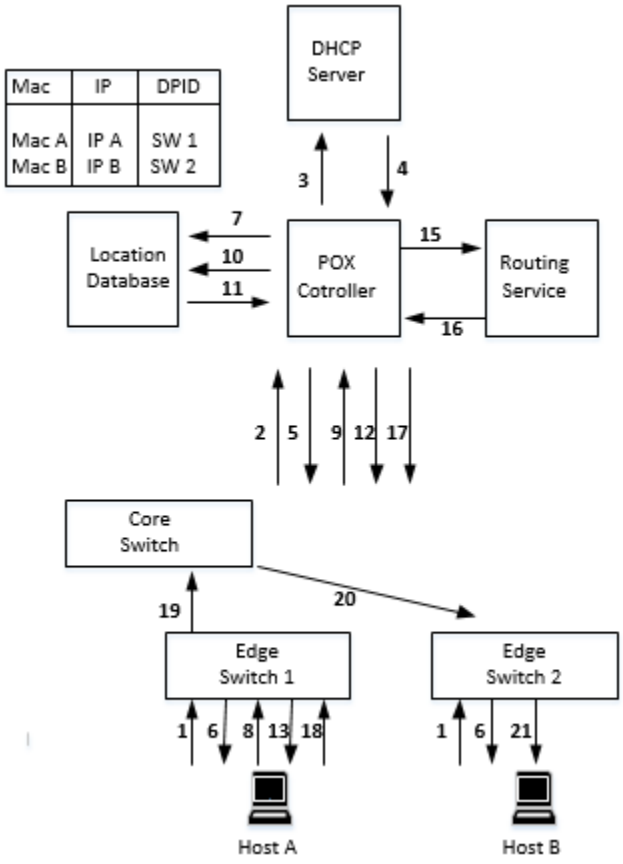

Figure 3.10: Packet forwarding between hosts on different edge switches.

8. When Host A communicates with Host B for the first time, Host A will send an ARP request.

9. Edge switch 1 intercepts the ARP request and sends it to the POX controller.

10. The POX controller will query the Location Database.

11. The Location Database will return the binding for Host B.

12. The POX controller will use the Mac address in the Host B binding to construct an ARP reply and forward it to edge switch 1.

13. Edge switch 1 forwards the reply to Host A.

14. The POX controller checks the DPID address in the Host B binding and determines that Host B is located on edge Switch 2.

15. The POX controller queries the Routing Service to determine how to route between edge Switch 1 and 2.

16. The Routing Service determines that there are two identical paths one through Core 1 and the other through Core 2. The Routing Service selects which path should be used and informs the POX controller. The path selection depends on the load balancing algorithm used. A detailed explanation of load balancing in OneSwitch is discussed in the next chapter.

17. The POX controller programs edge switch 1, and the selected core Switch with a flow entry for the selected path.

18. Host A forwards packets destined to Host B to edge switch 1.

19. Edge switch 1 forwards the packet to the selected core switch.

20. The selected core switch forwards the packets to edge switch 2.

21. Edge switch 2 forwards the packets to Host B.

3.3.3 Packet forwarding between mobile hosts

Figure 3.11 shows the steps for packet forwarding between Host A and Host B, when Host B moves from edge switch 1 to edge switch 2. The steps are as follows:

1. Host A on edge switch 1 communicates with Host B on edge switch 1. The steps are the same as outlined in section 3.3.1 above.

2. Host B on edge switch 1 moves to edge switch 2.

3. Edge switch 2 now receives traffic from Host B.

4. Since there are no entries programmed for this flow on edge switch 2, the flow is sent to the POX controller.

5. The POX controller records the source IP address, source Mac address, destination IP address, and destination Mac address. The POX controller uses that information to query the Location Database.

6. The Location Database returns entries for the source and destination of both IP and Mac addresses and their associated DPID addresses.

7. The POX controller realizes that Host B has moved from edge switch 1 to edge switch 2 and is still communicating with Host A. The POX controller updates the Location Database entry for Host B with the new DPID address and deletes the old entry on edge switch 1.

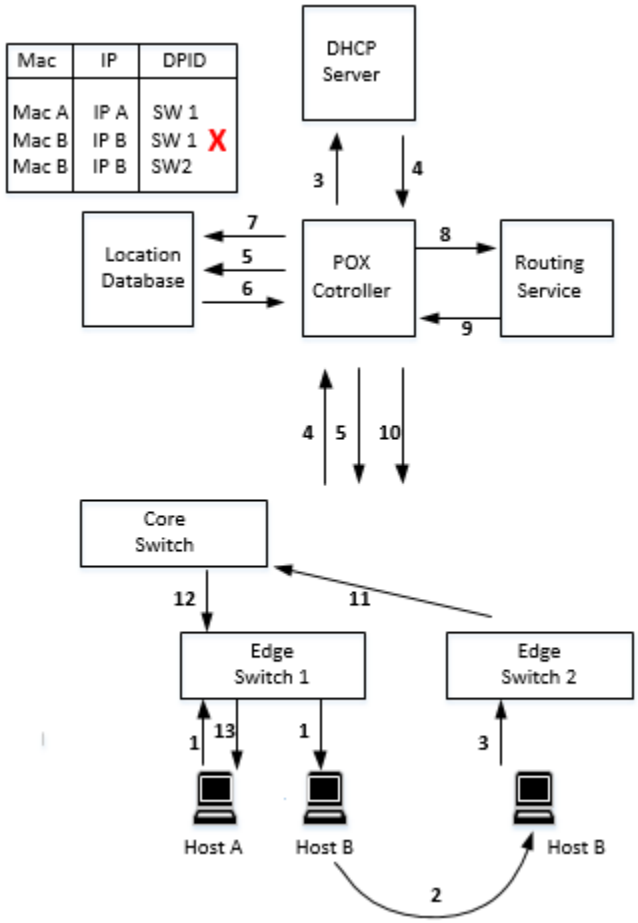

Figure 3.11: Packet forwarding between hosts on different edge switches.

8. The POX controller queries the Routing Service to determine how to route between Host B on edge switch 2 and Host A on edge switch 1 and vise versa. The Routing Service determines that there are two identical paths, one through core 1 and the other through core 2.

9. The Routing Service selects which path should be used and informs the POX controller. The path selection depends on the load balancing algorithm used. A detailed explanation of the routing algorithms used is discussed in the next chapters.

10. The POX controller programs edge switch 1, 2, and the selected core switch with a flow entry for the selected path.

11. Edge switch 2 forwards packets destined to Host A to the selected core switch.

12. The selected core switch forwards packets to edge switch 1.

13. Edge switch 1 forwards the packets to Host A. The same happens for traffic from Host A to Host B.

3.4 OneSwitch Features

The OneSwitch architecture provides a number of features that can be summarized as follows:

- The OneSwitch architecture provides simplified provisioning: The use of a DHCP server to assign hosts IP addresses from a single subnet greatly simplifies IP configuration. The use of the IP and Mac address to identify the host and the DPID address to identify the location of the host removes the need to divide hosts into subnets. This allows the ability to plug any device to any point on the network with minimal configuration.

- The OneSwitch architecture provides equidistant bandwidth and latency: Every Host connected to OneSwitch is two switches away from any other host. This provides predictable bandwidth and latency, which is important for application performance. It also simplifies the placement of hosts. A host can be connected to any edge switch without having any impact on application performance.

- The OneSwitch architecture uses DHCP and ARP requests to learn about hosts, therefore eliminating the need for broadcasts. This reduces unnecessary flooding allowing for greater scalability. It also eliminates the need for running a Spanning Tree Protocol, which allows efficient use of redundant paths.

- The OneSwitch architecture uses an external controller to perform perform packet forwarding decisions. This allows for backward compatibility with existing hardware and protocols.

- The OneSwitch architecture uses a Location Database that allows hosts to be mobile. A Host can move from one switch to another with minimal interruption.

- The OneSwitch architecture uses load balancing and flow optimization algorithms that use information gathered from the network to achieve high bisection bandwidth and low flow completions times between any two hosts communicating with each other. A detailed explanation of these algorithms is presented in the next chapters.

Chapter 4

Load Balancing in OneSwitch

Preserving packet order while achieving efficient load balancing is a difficult problem to solve, especially when the traffic demand matrix is unknown and traffic patterns exhibit a large degree of variation; unfortunately this is the case with many data centers today [46].

Randomized routing algorithms, such as ECMP, try to achieve optimal load balancing for most flows instead of optimizing load balancing for all flows. To demonstrate this we formulate randomly assigning m flows to n links, as a Ball and Bins problem [3]. The balls here represent flows, while the bins represent links.

4.1 Balls and Bins Model

In the Balls and Bins model, each time we have a ball, we pick one bin independently and uniformly at random, and then throw the ball in that bin, which implies that the probability that a ball falls into any given bin is $1/n$. Based on this process, the Balls and Bins model allows us to answer a number of questions, that can help us better understand the performance of randomized load balancing algorithms.

Question 1: What is the probability of any two balls falling into one bin?

We can view this question as after the first ball falls into a particular bin p, what is the probability of the second ball falling into bin p. Since balls falling into any bin is equally likely and tosses are independent of each other, the probability of the second ball falling into bin p is simply $1/n$. Thus the probability of any two balls falling into one bin is $1/n$. We can use Bayes Rule to prove the correctness of this probability, which gives

$$\begin{aligned} Pr[\varepsilon_{12}] &= \sum_{p=1}^{n} Pr[\eta_2^p|\eta_1^p] Pr[\eta_1^p] \\ &= \sum_{p=1}^{n} \frac{1}{n} Pr[\eta_1^p] \\ &= \frac{1}{n} \end{aligned} \tag{4.1}$$

Question 2: What is the expected number of collisions when we toss m balls?

To answer this question, we use a indicator random variable X_{ij} to indicate if an event happens or not. The random variable $X_{ij} = 1$ if a collision happens and $X_{ij} = 0$ if a collision doesn't happen. Since X_{ij} only takes zero or one as its value, they are Bernoulli variables,

and tosses can be considered as a sequence of Bernoulli trials with a probability $1/n$ of success. We also define X to be the number of collisions, i.e. Then we calculate $E[X]$ as follows:

$$\begin{aligned} E[x] &= \sum_{i \neq j} E[X_{ij}] \\ &= \sum_{i \neq j} Pr[x_{ij} = 1] \\ &= \frac{1}{n} \binom{m}{n} \end{aligned} \tag{4.2}$$

From now on, we assume $m = n$ for simplicity. All the analysis can be generalized to $m \neq n$.

Question 3: What is the probability of a particular bin being empty?

The probability of a ball not fall into a particular bin is $1 - \frac{1}{n}$. Thus, we have:

$$Pr[\text{bin i is empty}] = (1 - \frac{1}{n})^n \to \frac{1}{e} \tag{4.3}$$

Question 4: What is the probability of a particular bin having k balls?

$$\begin{aligned} Pr[\text{bin i has k balls}] &= \binom{n}{k} (\frac{1}{n})^k (1 - \frac{1}{n})^{n-k} \\ Pr[\text{bin i has k balls}] &\leq \frac{n^k}{k!} \frac{1}{n^k} = \frac{1}{k!} \end{aligned} \tag{4.4}$$

Question 5: What is the probability of a particular bin having at least k balls?

If we look at any subset of balls of size k, then the probability that the subset of balls fall into bin i is $\frac{1}{n^k}$. Note that we no longer have the $(1 - \frac{1}{n})^{n-k}$ factor, because we don't care

about where the rest of the balls fall. We then take a union bound of these probabilities over all $\binom{n}{k}$ subsets of size k. The events we are summing over, though, are not disjoint. Therefore, we can only show that the probability of a bin having at least k balls is at most $\binom{n}{k}(\frac{1}{n})^k$.

$$Pr[\text{bin i has at least k balls}] \leq \binom{n}{k}(\frac{1}{n})^k$$

Using Stirling's approximation, we have

$$Pr[\text{bin i has at least k balls}] \leq (\frac{e}{k})^k \tag{4.5}$$

If we view question 5 as a load balancing problem, then we are interested in finding some k so that $(\frac{e}{k})^k$ is very small. The solution is $k = O(\frac{\ln n}{\ln \ln n})$.

Theorem 4.1.1. *With high probability, i.e.* $1 - \frac{1}{n}$*, all bins have at most* $\frac{3 \ln n}{\ln \ln n}$ *balls.*

Proof. let $k = \frac{3\ln n}{\ln\ln n}$ from 4.5, we have

$$\begin{aligned} Pr[\text{bin i has at least k balls}] &\leq (\frac{e}{k})^k = (\frac{e\ln\ln n}{3\ln n})^{\frac{3\ln n}{\ln\ln n}} \\ &\leq \exp(\frac{3\ln n}{\ln\ln n}(\ln\ln\ln n - \ln\ln n)) \\ &\leq \exp(-3\ln n + \frac{3\ln n\ln\ln\ln n}{\ln lnn}) \end{aligned}$$

When n is large enough

$$Pr[\text{bin i has at least k balls}] \leq exp(-2\ln n) = \frac{1}{n^2}$$

Using union bound, we have

$$Pr[\text{any bin has at least k balls}] \leq n\frac{1}{n^2} = \frac{1}{n}$$

which implies

$$Pr[\text{all bins have at most k balls}] \leq 1 - \frac{1}{n}$$

□

Theorem 4.1.1 shows that if we distribute balls independently and uniformly at random, the largest number of balls in any bin is approximately $\frac{3\ln n}{\ln\ln n}$ with high probability.

4.2 The Power of Two Choices

In the Balls and Bins model, each time we have a ball, we pick one bin independently and uniformly at random, and then throw the ball in that bin. Azar et al [17] show that if instead we pick two bins independently and uniformly at random, and put a ball into the bin with

fewer balls, now the bins will have at most $\frac{\ln \ln n}{\ln 2}$ balls. By increasing the choices of bins for the balls to be thrown in from one to two random choices, the maximum number of balls in a bin is reduced by an exponential factor. This phenomenon is called the power of two random choices. Increasing the choices to more than two will improve the maximum load by a constant factor $\frac{\ln \ln n}{\ln d}$, where d is the number of choices and is ≥ 2.

4.3 Supermarket Model

The Balls and Bins model is a static model, the number of balls is fixed. To capture more realistic settings, where balls arrive and leave over time, a dynamic model is needed. In looking for a dynamic generalization of Balls and Bins problems, the Supermarket model can be considered [21].

In the Supermarket model, balls arrive as a Poisson stream of rate λ, where $\lambda \leq 1$, at a collection of n bins. Each ball chooses d bins independently and uniformly at random, and is assigned to the bin with the lowest number of balls. Balls are served according to First in First out (FIFO) protocol, and the service time for a ball is exponentially distributed with mean 1.

In this model, it is assumed that the time to obtain information about bin lengths and the time to move to a bin is zero. Also, it is assumed that that the service rates are the same for all bins. The Supermarket model can be easily analyzed when $d = 1$. In this case, the arrival stream can be split into independent Poisson streams for each bin, and hence each

bin acts as a simple $M/M/1$ queue. For $d \geq 2$, knowing the length of one queue affects the distribution of the length of all the other queues, hence the Supermarket model for $d \geq 2$ is a Markov process. The Markovian nature of the Supermarket model allows sophisticated techniques to be applied that can be used to analyze stability and convergence of the system. These techniques can be used to give an alternative proof of the $\frac{\ln \ln n}{\ln 2}$ upper bound for the maximum load that was derived from the static Balls and Bins model.

The Markov process depends on the assumptions of Poisson arrivals and exponential service times. The author in [21] shows how to approximate non- Markovian models with Markovian models, allowing use of Markovian techniques to non-Markovian systems.

To understand the time evolution of the Supermarket model let $n_i(t)$ be the number of queues with i customers at time t, $m_i(t)$ to be the number of queues with at least i customers at time t, $p_i(t) = \frac{n_i(t)}{n}$ to be the fraction of queues of size i, and $s_i(t) = \sum_{k=i}^{\infty} p_i(t) = \frac{m_i(t)}{n}$ to be the tails of $p_i(t)$. from here on the reference to t is dropped in the the notation for simplification.

In an empty system, which corresponds to one with no customers, s_0=1 and s_i=0 for $i \geq 1$.

The the state of the system can be represented at any given time by an infinite dimensional vector $\vec{s} = (s_0, s_1, s_2....)$. The time evolution of the system is specified by the following set of differential equations:

$$\begin{cases} \frac{ds_i}{dt} = \lambda(s_{i-1}^d - s_i^d) - (s_i - s_{i+1}) \\ \\ s_0 = 1 \end{cases} \tag{4.6}$$

setting $\frac{ds_i}{dt}$=0 leads to the system represented by 4.6, with $d \geq 2$ having a unique fixed point given by:

$$s_i = \lambda^{\frac{d^i-1}{d-1}} \tag{4.7}$$

The expected time a customer spends in the Supermarket system for $d \geq 2$ is given by:

$$T_d(\lambda) = \sum_{i=1}^{\infty} \lambda^{\frac{d^i-1}{d-1}} \tag{4.8}$$

4.4 Formulating load balancing in OneSwitch as a Supermarket Problem

We formulate routing flows in OneSwitch as a Supermarket problem. The OneSwitch architecture, as explained in the previous chapter, consists of identical switches each with n uplinks. Each switch sends flow routing decisions to a centralized POX controller. The POX controller, with the assistance of the Routing Service, then selects $d \leq n$ links uniformly at random and sends d queries to the edge switches inquiring about the load of the selected links. When the centralized POX controller obtains all d answers from an edge switch, it selects the uplink with the least load (ties are broken arbitrarily) and routes the flow through that link.

Our goal of formulating load balancing in OneSwitch as a Supermarket problem is to achieve better performance than randomized algorithms, such as ECMP. As mentioned previously choosing $d = 2$ reduces the maximum load by an exponential factor and choosing $d > 2$

further improves the maximum load by a constant factor. An important question that comes to mind is what is the optimal value of d that achieves a better performance than ECMP but at the same time is practical. For example, querying all d uplinks of an edge switch that has 64 uplinks might yield worse performance than ECMP, because the POX controller has to wait until it receives the results for all queries to and then process all the results before selecting an uplink.

In the Supermarket model the time to obtain information about bin lengths and the time to move to a bin is assumed to be zero. In practical implementations this is not the case. For example, in OneSwitch routing decisions can not be made until the Edge switch sends the flow to the POX controller and then the POX controller queries the edge switch uplinks and receives an answer.

To derive an optimal value for d we follow a similar approach to [56] and capture the time it takes to obtain load information. Let m denote the mean time to process a monitoring request. The ratio $L = \frac{\mu}{m}$ quantifies the impact that load requests have on the actual flow completion time. The POX controller routes incoming flows according to a First-Come-First-Served (FCFS) policy. Flows arrive at the POX controller as a Poisson stream of rate λ, where $0 \leq \lambda \leq 1$ and the service time is exponentially distributed with mean service time $\frac{1}{\mu} = 1$. Each flow creates d load queries. Since we have n uplinks and they are chosen uniformly at random, each uplink gets a Poisson load query stream with incoming rate of $\lambda \cdot d$. Since the mean service time is 1, and the mean service time for load requests is L, the

effective service rate is:

$$\mu' = 1 - \lambda \cdot d \cdot L \tag{4.9}$$

Let $\rho = \frac{\lambda}{\mu'}$ be the arrival rate normalized by the effective service rate; we get:

$$\rho = \frac{\lambda}{1 - \lambda \cdot d \cdot L} \tag{4.10}$$

We observe that ρ must be smaller than 1 in order to keep the system stable. In other words, if we want to have a stable state where the uplinks in the system have finite flows, we must have $\rho < 1$, or

$$\lambda < \frac{1}{1 + d \cdot L} \tag{4.11}$$

Note that Equation 4.11 indicates that in some cases where the flow arrival rate is high, the time for the load query can cause the system to become unstable. This means that in some cases we would be better off choosing a smaller value of d or even performing a random uplink selection, similar to ECMP.

To calculate the expected time a flow spends in the system we define $n_i(t)$ to be the number of links with exactly i flows and $s_i(t)$ as the fraction of the links with at least i flows.

$$s_i(t) = \sum_{k=1}^{\infty} \frac{n_k(t)}{n} \tag{4.12}$$

We can see that $s0 = 1$ and $s1$ is the fraction of non empty servers. The average queue length at time t is defined as:

$$\sum_{n=1}^{\infty} s_i(t) = \frac{1}{n} \sum_{n=1}^{\infty} i \cdot n_i(t) \tag{4.13}$$

For any $d > 1$, L, finite n, and λ that satisfy Equation 4.13, the system is in a stable state, and when t is large enough there is a fixed probability to be in each of the states defined by the vector $\vec{s} = (s_0, s_1, s_2....)$.

In such a case, a new flow is routed on a link with a load i only if all d chosen links have queues not smaller than i, and at least one of them has a queue of size i. This happens with probability $s_i^d - s_{i+1}^d$. Similarly, the probability that a flow finishes at a link with a load of i is $s_i - s_{i+1}$.

This implies that the following differential equation holds for $i \geq 1$:

$$\frac{ds_i}{dt} = \rho(s_{i-1}^d - s_i^d) - (s_i - s_{i+1}) \tag{4.14}$$

where $s_0 = 1$. The set equation has the following unique fixed point [21].

$$s_i = \rho^{\frac{d^i - 1}{d-1}} \tag{4.15}$$

In order to compute the expected time a flow spends in the system in units of the service rate, we divide the expected queue length by λ.

$$\begin{aligned} T_d(\lambda) &= \frac{1}{\lambda} \sum_{i=1}^{\infty} \rho^{\frac{d^i - 1}{d-1}} \\ &= \frac{1}{1 - \lambda \cdot d \cdot L} \sum_{i=1}^{\infty} \rho^{\frac{d^i - 1}{d-1} - 1} \\ &= \frac{1}{1 - \lambda \cdot d \cdot L} \sum_{i=1}^{\infty} \rho^{\frac{d^i - d}{d-1}} \end{aligned} \tag{4.16}$$

$T_d(\lambda)$ is a monotonically decreasing function for any $0 < \lambda < 1$. As d increases $T_d(\lambda)$ decreases, resulting in smaller waiting times for flows. At some point as d continues to

increase $T_d(\lambda)$ starts increasing, resulting in higher waiting times, and when $\lambda \cdot d \cdot L$ approaches 1, the waiting time in the system goes to infinity.

Figures 4.1, 4.2, 4.3, 4.4 show the average waiting time for flows versus the number of queried uplinks for different values of λ and L. One can see that for low values of λ and L the value of d that causes the system to become unstable is high. For example, in the case where $\lambda = 0.55$ and $L = 0.01$ the value of d that causes the system to become unstable is around $d = 80$. On the other hand for higher values of λ and L only a small number of queried uplinks can cause the system to become unstable. For example, in the case where $\lambda = 0.85$ and $L = 0.1$ the value of d that causes the system to become unstable is very low, $d = 2$. In such cases it's better not to query any uplinks, because of the high cost incurred.

In order to derive the optimal value of d for a fixed set of parameters, we need to obtain the derivative of equation 4.16 with respect to d. Since 4.16 has no closed formulation, we numerically find the optimal values of d as a function of the system load λ for different values of L. Table 4.1 shows d and $T_d(\lambda)$ for different values of λ and L. The highlighted values of d represent the optimal d for a specifc vlaue of λ and L. The optimal value of d is the value with the smallest waiting time, right before $T_d(\lambda)$ starts to increase.

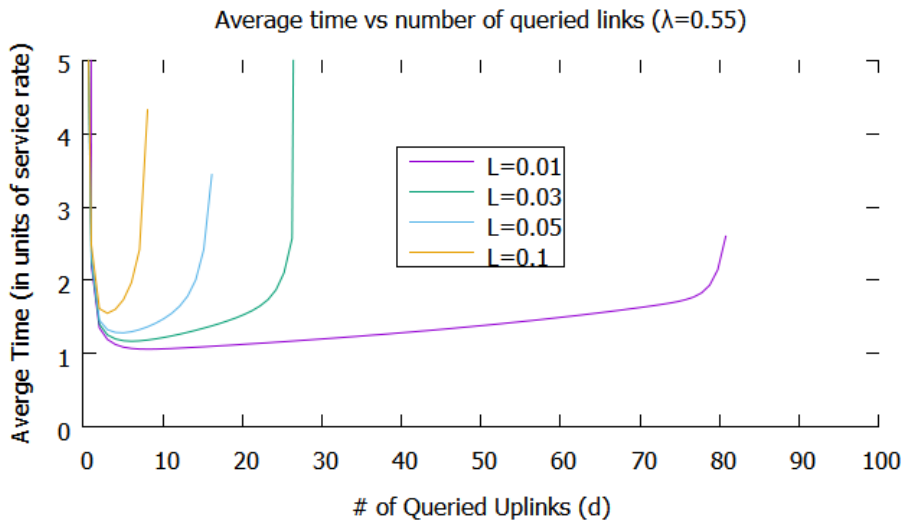

Figure 4.1: Average time vs number of queried uplinks for $\lambda = 0.55$ and $L = 0.01, 0.03, 0.05$ and 0.1.

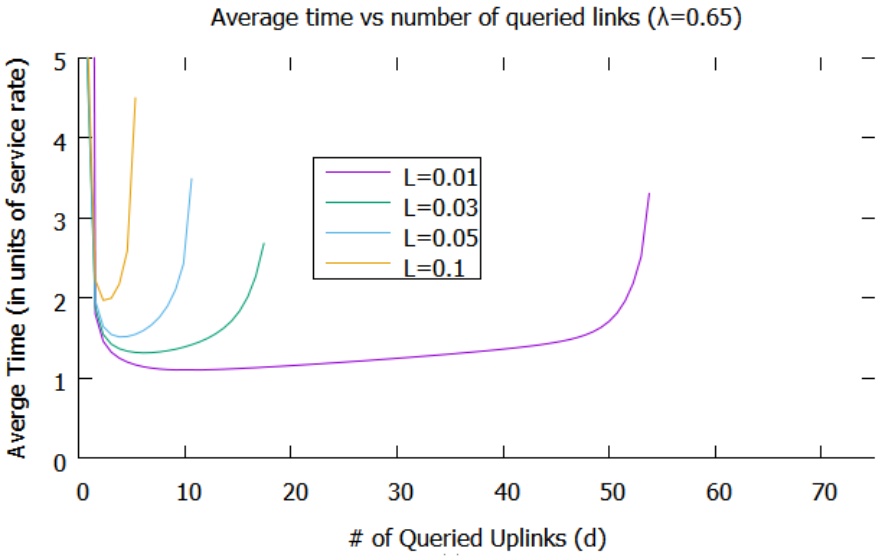

Figure 4.2: Average time vs number of queried uplinks for $\lambda = 0.65$ and $L = 0.01, 0.03, 0.05$ and 0.1.

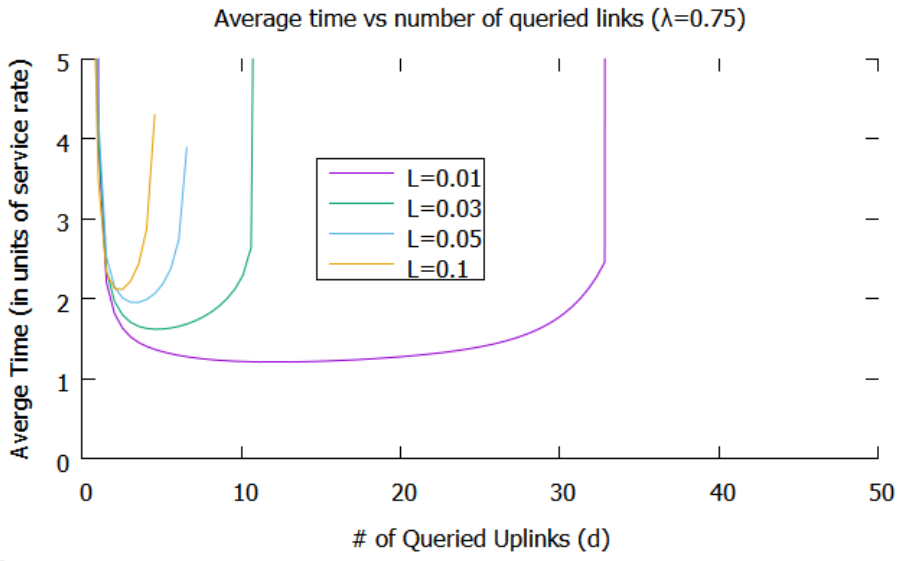

Figure 4.3: Average time vs number of queried uplinks for $\lambda = 0.75$ and $L = 0.01, 0.03, 0.05$ and 0.1.

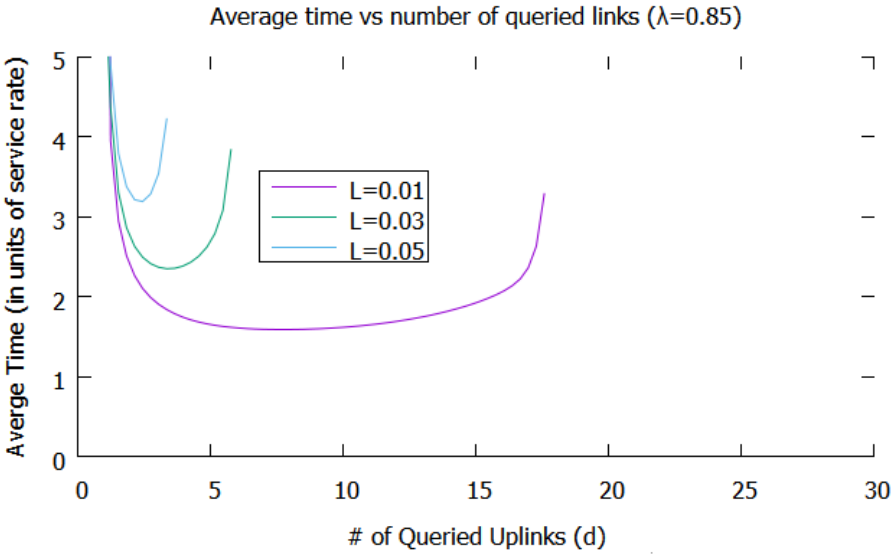

Figure 4.4: Average time vs number of queried uplinks for $\lambda = 0.75$ and $L = 0.01, 0.03$ and 0.05.

Table 4.1: d and $T_d(\lambda)$ for different values of λ and L

λ	L	d	$T_d(\lambda)$	λ	L	d	$T_d(\lambda)$	λ	L	d	$T_d(\lambda)$
0.55	0.01	7	1.06	0.65	0.03	6	1.31	0.75	0.05	4	1.98
0.55	***0.01***	***8***	**1.05**	***0.65***	***0.03***	***7***	**1.31**	***0.75***	***0.1***	***2***	**2.8**
0.55	0.01	10	1.06	0.65	0.03	8	1.33	0.75	0.1	3	3.71
0.55	0.03	4	1.19	0.65	0.05	3	1.54	0.85	0.01	7	1.59
0.55	***0.03***	***6***	**1.16**	***0.65***	***0.05***	***4***	**1.51**	***0.85***	***0.01***	***8***	**1.58**
0.55	0.03	7	1.17	0.65	0.05	5	1.53	0.85	0.01	9	1.59
0.55	0.05	4	1.28	***0.65***	***0.1***	***2***	**1.99**	0.85	0.03	2	2.71
0.55	***0.05***	***5***	**1.28**	0.65	0.1	3	2.01	***0.85***	***0.03***	***3***	**2.37**
0.55	0.05	6	1.29	0.65	0.3	2	∞	0.85	0.03	4	2.38
0.55	0.1	2	1.61	0.75	0.01	8	1.23	***0.85***	***0.05***	***2***	**3.25**
0.55	***0.1***	***4***	**1.6**	***0.75***	***0.01***	***9***	**1.22**	0.85	0.05	3	3.5
0.55	0.1	5	1.72	0.75	0.01	10	1.21	0.95	0.01	3	3.27
0.55	***0.3***	***2***	**3.01**	0.75	0.03	4	1.62	***0.95***	***0.01***	***4***	**3.21**
0.55	0.3	3	∞	***0.75***	***0.03***	***5***	**1.61**	0.95	0.01	5	3.89
0.65	0.01	7	1.11	0.75	0.03	6	1.65	0.95	0.03	2	∞
0.65	***0.01***	***8***	**1.06**	0.75	0.05	2	2.15	0.97	0.03	2	∞
0.65	0.01	9	1.09	***0.75***	***0.05***	***3***	**1.95**	0.99	0.01	2	∞

Figure 4.5: 104-bit 5-Tuple Flow ID.

4.5 Selective Randomized Load Balancing (SRL)

We use the Supermarket formulation discussed in the previous section to design SRL, shown in Algorithm 1. SRL is used as part of the Routing Services component in OneSwitch. When packets of a new flow arrive at an edge switch, they are sent to the POX controller, which forwards it to SRL. SRL will use the values of λ and L, discussed in previous section, to derive the optimal value of d. SRL will then randomly pick d links out of n links. Each link is selected by applying a CRC 32 hash [18] function to the 5-tuple flow ID, shown in Figure 4.5, then adding an offset value, o, and performing a modulo n operation, where n is the number of links. The offset value is different for each link, and guarantees that d different links are selected for each flow ID. For example, if d=3, then offset values, could be chosen as 1,2, and 3 in order to select 3 different links. The selection function is expressed as follows:

$$S = (CRC32(5 - Tuple) + o)Mod(n) \tag{4.17}$$

SRL queries the selected links for load statistic and chooses the least loaded link. SRL then programs the flow table on the edge switch with an entry that routes all packets belonging to the flow across the chosen link. Figure 4.6 shows how SRL performs link selection. If the flow is idle for more than the idle timeout period, the expired flow entry is flushed. The default timeout period is 60 seconds. If the flow reappears then the process is repeated again.

Algorithm 1 SRL

procedure AT TIME T=t_0 POLL SWITCHES FOR FLOW SIZES

$F_{t0} = FlowSize\{flow\}$, d=1, $Speed = 1Gbps$, $tm = 1$ Second

if $Flow$ received **then**

Query d links for load statistics and select least loaded

Q_t= Load Query Time

$I = Flow\_Inter\_Arrival\_Times\{flow\}$

Route Flow over selected link

At time T=t_m poll switches for flow sizes

$F_{tm} = FlowSize\{flow\}$

for all $flow$ received **do**

$I_{avg} = sum(I)/len(I)$

$\lambda = 1/I_{avg}$

for all $flow$ in F_{t0} and Not in F_{tm} **do**

$F_C = Flow\_Completed\{flow\}$

$T_f = Flow\_Comp\_Times\{flow/Speed\}$

$Avg\_Comp\_Time = sum(T)/len(T)$

$L = Q_t/Avg\_Comp\_Time$

Use λ and L to compute d from look-up table

$F_{t0} = F_{tm}$

Repeat Every T=t_m

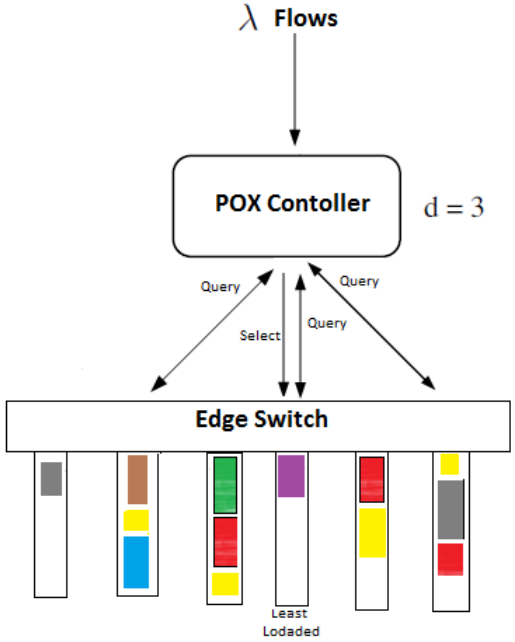

Figure 4.6: SRL link selection.

We now turn to discussing some practical considerations with SRL. The first is determining the optimal value of d. Because there is no efficient way of computing the value of d, we pre-compute the optimal value of d for several values of λ and L, and store them in a lookup table that can be accessed by SRL. Table 4.2 shows the optimal d for different values of λ and L.

Another practical issue is actually determining the values of λ and L. These value are not constant, they change over time, and they are unknown, therefore they have to be dynamically estimated.

λ represents the average flow arrival rate, which is the inverse of the average inter arrival rate

Table 4.2: SRL d lookup table

λ Range	L Range	Optimum d
0-0.55	0-0.3	10
0-0.55	>0.3	2
0.55-0.75	0-0.03	7
0.55-0.75	0.03-0.05	5
0.55-0.75	0.05-0.1	4
0.55-0.75	0.1-0.3	2
0.55-0.75	>0.3	NA
0.75-0.85	0-0.03	5
0.75-0.85	0.03-0.05	4
0.75-0.85	0.05-0.1	3
0.75-0.85	0.1-0.3	2
0.75-0.85	>0.3	NA
0.85-0.95	0-0.03	2
0.85-0.95	>0.03	NA

of flows. Each time a new flow is received by an edge switch an Openflow Packet_in message is sent to the POX controller. SRL records the time of Packet_in messages and calculates the average inter arrival rate by subtracting consecutive flow arrival times and averaging them over a one second time period. The average inter arrival rate for the previous time period is used to estimate λ for the current time period. We believe a time period of one second is reasonable to capture a large enough number of flows, because of the relatively large round trip times, around 10-20 millisecond, of Packet_in messages to the controller [69].

L reflects the ratio between time it takes the POX controller to receive an answer to a load query and the average flow completion time.

The time to receive an answer for a load query can be easily computed by the POX controller, this is not the case for average flow completion times. The POX controller does not know the actual flow completion time of flows, only the switch that's routing the flows has this information. In addition, this information needs to be available to the POX controller when it needs to make a load query. We solve this problem by obtaining the flow completion time from the switch periodically and using the information when a load query has to be made.

SRL periodically polls switches for flow statistics every 1 second. The number of flows and their size is recorded and compared to the previous period to determine which flows have completed. Flows that have completed are the ones that were present in a previous polling cycle, but are no longer available in the current cycle. The sizes of completed flows is used to estimate the average flow completion time. For example if a previously polled 1Gbps link had 3 flows, $75MB$, $50MB$, $25MB$, and during the current polling cycle those flows are no

longer available, then the flow completion times would be $\frac{75MB}{1Gbps} = 75ms$, $\frac{50MB}{1Gbps} = 50ms$, $\frac{25MB}{1Gbps} = 25ms$. The average flow completion time is $\frac{75+50+25}{3} = 50ms$.

The values of λ and L also directly impact the performance of the controller itself. The higher the values of λ and L the higher the cpu utilization. This can be easily verified using the get *cputimes*() method from *psutil*, a cross platform process and system utilities module for Python [120]. The performance of the controller is a key factor for network scalability [84]. A detailed analysis of the controller performance is presented in 6.1.

4.6 SRL Simulation

We simulate a large data center with $K = 72$ switches and 1152 hosts. We analyze the performance of ECMP and SRL, in terms of flow collisions, by performing a number of simulation scenarios. For each scenario we perform ECMP and SRL load balancing under three parametric classes:

1. Flow traffic patterns.
2. Number of flows.
3. Flow size.

In terms of the flow traffic pattern, we simulate hosts sending flows to other hosts according to the following patterns:

1. Step(i) : host with index x sends flows to host with index $(x + i)$ $MOD(h)$ (h is the number of Hosts).

2. Random: Each host randomly sends flows to any other host switch with uniform probability.

3. Shuffle: Iterates through hosts, with each host sending flows to any of the other available hosts at random.

In terms of the number of flows, we simulate with three different number of flows:

1. 10 flows

2. 100 flows

3. 1000 flows

In terms of flow size distribution, we test with three flow size distributions:

1. Uniform - All flows have the same size.

2. Predominately elephant - Elephant flows occupy 75 percent of the total bandwidth, and the mice flows occupy the remaining bandwidth.

3. Predominately mice - Mice flows will occupy 75 percent of the total bandwidth, and the elephant flows will occupy the remaining bandwidth.

All possible combinations of the three parametric classes are tested.

Figures 4.7, 4.8, and 4.9 show the number of hash collisions normalized to number of flows for ECMP and SRL for 10, 100, 1000 flows using a Step (1), Shuffle, and Random traffic pattern with a uniform, predominately mice, and predominately elephant flow size distributions.

As expected, we observe that the scenario with predominately elephant flows has the most hash collision when using ECMP then followed by uniform flows and predominately mice flows respectively.

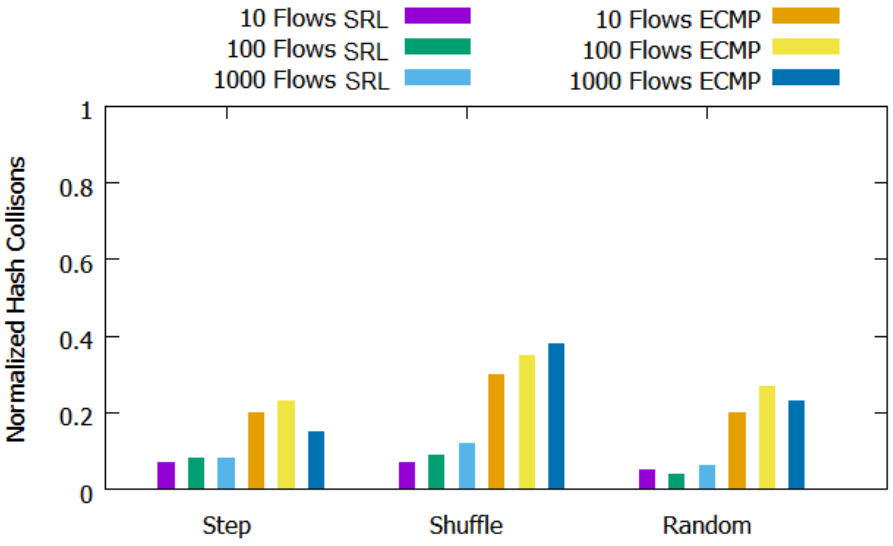

Figure 4.7: Normalized hash collision-Step/Uniform.

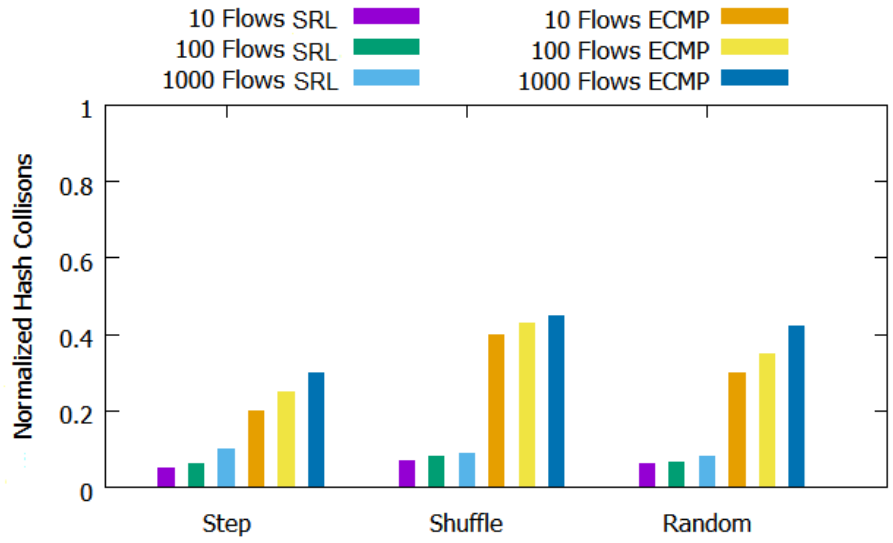

Figure 4.8: Normalized hash collisions-Shuffle/Predominately mice.

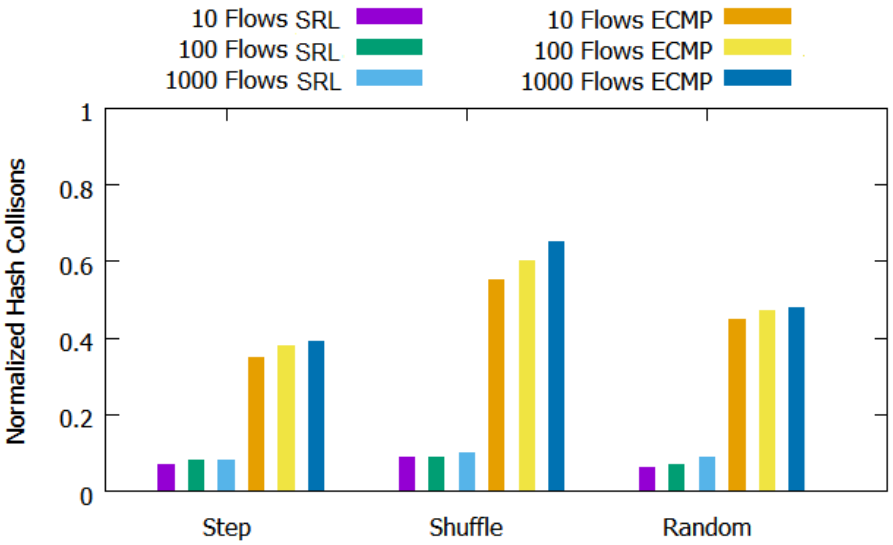

Figure 4.9: Normalized hash Collisions-Random/Predominately elephant.

Chapter 5

Flow Optimization in OneSwitch

An optimization problem is the problem of finding the best solution from all feasible solutions. Linear Programming is a category of optimization problems that deals with the optimization of systems where different activities compete for a set of resources and the relations of the activities can be expressed mathematically by linear functions. Integer Programming, in which some or all of the variables are restricted to be integers, is considered as an extension of Linear Programming that allows the mathematical modelling of a broader type of decision problems.

Linear Programming is an established discipline, its methods have been successfully applied in a large number of practical problems and there are robust and efficient software implementations. However, Integer Programming is still a challenging field of research. There are no known polynomial algorithms to solve as opposed to the general Linear Program-

ming problem. Although several important specific problems have been studied for several decades, and very significant progresses have been made in their resolution, they remain difficult to solve. Some of them have a combinatorial structure, that is, the set of feasible solutions is a set of objects that can be explicitly numerated, but their number is too large for the enumeration to be efficient in a solution procedure. Several approaches, such as Cutting Plane [16], Branch-and-Bound [30], Branch-and-Cut [11], and branch-and-price [13] have been developed to tackle this inherent complexity of Integer Programming.

5.1 Multicommodity Flow Problem

A Multicommodity Flow Problem is a Linear Programing optimization problem that involves routing multiple commodities across a shared network. All commodities must be routed from their origins to their destinations at a minimum cost, in a network with capacitated arcs. The general Linear Programming model for the Multicommodity Flow Problem is composed of two sets of constraints, flow conservation of the commodities and capacities of the arcs.

5.1.1 Integer Multicommodity Flow Problem

The integer minimum cost Multicomodity Flow Problem is defined over a directed network in which several commodities share the capacity of the arcs, in order to be shipped from their origin to their destination nodes. Associated with each arc of the network and with each commodity there is a unit flow cost. The minimum cost integer Multicommodity Flow

Problem amounts to finding the minimum cost routing of all the commodities, taking into account that each unit of each commodity cannot be split. The solution to the integer Multicommodity Flow Problem is NP-hard [4].

5.1.2 Binary Multicommodity Flow Problem

The binary minimum cost Multicommodity Flow Problem is defined over a directed network in which several commodities share the capacity of the arcs in order to be shipped from the origin to the destination nodes. There is a unit cost flow associated with each arc of the network and with each commodity. The minimum cost binary Multicomodity Flow Problem amounts to finding the minimum cost routing of all the commodities, taking into account that the flow of each commodity cannot be split. The solution to the Binary Multicommodity Flow Problem is NP-Complete [4].

5.2 Formulating routing in OneSwitch as a Binary Multicommodity Flow Problem

In this section, we formulate routing in OneSwitch as a binary Multicommodity Flow Problem and describe it's properties. The problem to be solved, can be stated succinctly as follows: how does one optimally assign flows (commodities) to the available paths in OneSwitch such that link bandwidths are efficiently used, links are not over subscribed, and flows are

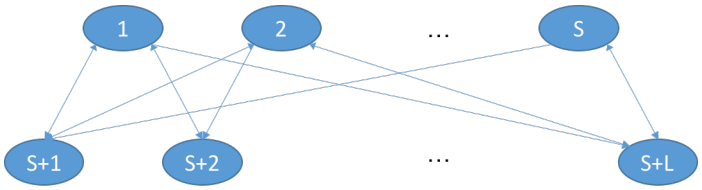

Figure 5.1: OneSwitch network represented as a directed graph.

not split among different paths. If flows were allowed to be split among different paths, this becomes an integer Multicommodity Flow Problem.

In order to formally define the binary Multicommodity Flow Problem, we first describe directed graphs and related definitions.

A directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{A})$ consists of $\mathcal{N}$ nodes and a set $\mathcal{L}$ of pairs of distinct nodes from $\mathcal{N}$ called Arcs (Edges). An Arc (i, j) consists of a directed connection from outgoing node i into incoming node j. In the context of OneSwitch architecture, the nodes $\mathcal{N}$ represent the switches through which data flows, and the arc's $\mathcal{A}$ represent the physical connections between switches. Figure 5.1 shows the data plane of OneSwitch represented as directed graph. The directed graph has S core nodes and L edge nodes, with each core node $S_i, \forall i = 1, ..., S$ connected bidirectionally to every edge node $L_j, \forall j = 1, ..., L$. The bidirectional connects signify that traffic can flow in both directions between a core node and a edge node.

We use the directed graph in Figure 5.1 to cast routing in OneSwitch as an optimization problem as follows: We are given a set K of h flows indexed by k. A pair of indices ij represent a link from origin node i to destination node j. Each flow k is characterized by an

origin o^k and a destination d^k, and a required bandwidth r^k. We also define the capacity of the link ij with u_{ij} and a cost c_{ij}^k associated with the k^{th} flow on link ij, with the assumption that $c_{ij}^k \geq 0, \forall ij \in A, \forall k \in K$. The routing objective then, is to assign each flow k to a certain path (a series of links) such that the following constraints are met:

1. Flow rates are conserved.

2. The available capacities of the links are not exceeded.

3. Each flow rate is indivisible.

Mathematically, we can express the objective function with equation 5.1.

$$min \sum_{k \in K} \sum_{ij \in \mathcal{A}} c_{ij}^k r^k x_{ij}^k \tag{5.1}$$

The flow rate constraint can be expressed mathematically as:

$$\sum_{j:ij \in \mathcal{A}} x_{ij}^k - \sum_{j:ji \in \mathcal{A}} x_{ji}^k = \begin{cases} 1 \ i = o^k \\ -1 \ i = d^k \\ 0 \ i \neq o^k, \ i \neq d^k \end{cases} \tag{5.2}$$

The link capacity constraint is captured by

$$\sum_{k \in K} r^k x_{ij}^k \leq u_{ij}, \forall \ ij \in \mathcal{A} \tag{5.3}$$

Finally, the indivisibility of the commodity rate is captured by the equation:

$$x_{ij}^k \in \{0, 1\}, \ \forall k \in K, \ \forall ij \in \mathcal{A} \tag{5.4}$$

The mathematical formulation given by equations 5.1, 5.2, 5.3, and 5.4 allow us to explore two relevant issues:

1. Achieving better link utilization.

2. Eliminating or reducing hash collisions

Equation 5.1 maps directly to achieving better link utilization, because it seeks to minimize the total amount of bandwidth on each link. In our scenario, $c_{ij}^{k} = 1, \forall ij \in A, \forall k \in K$ because we treat all flows with equal priority.

Equation 5.2 forces the edge nodes to be bandwidth sources or sinks (1 or -1), and it forces core nodes to pass all information through without being a source nor a sink. This insures that the optimization algorithm does not come up with a trivial solution of not assigning any flows to any links.

Equation 5.3 eliminates hash collisions because it does not allow the combination of flows assigned to a link to exceed it's capacity.

Finally, the combination of equations 5.3 and 5.4 insure the indivisibility of flows.

The above formulation leads to one decision variable being used for each flow and link, one constraint for each flow and switch, and one constraint for each capacitated link. These decision variables and constraints can be very large. For example, a OneSwitch data plane that has a total of 60 switches, 20 edge switches and 40 core switches, that has to route 1000 flows will have 800,0000 decision variables representing flows and links, 60000 constraints

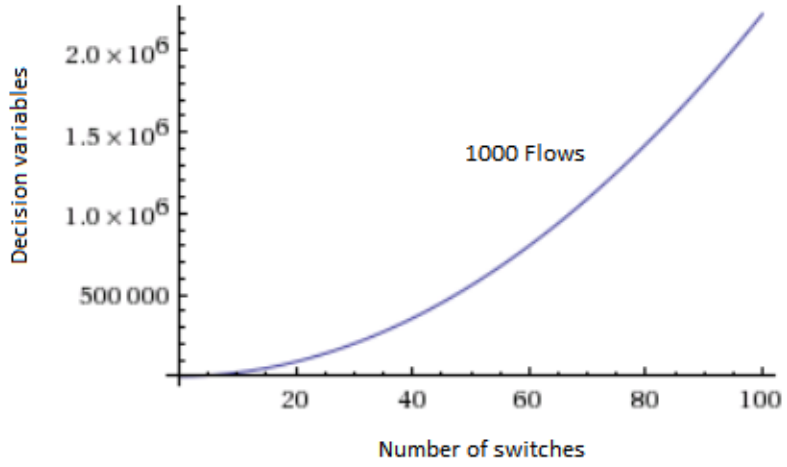

Figure 5.2: Number of decision variables versus the total number of switches for 1000 flows.

representing flows and switches, and 800 constraints representing the capaicitated links. Figure 5.2 shows the number of decision variables versus the total number of switches for a OneSwitch network with 1000 flows.

One might wonder if a solution to such a model is solvable in polynomial time. The general binary integer Multicommodity Flow Problem is NP-complete, but this is usually because the details of the network architecture is hidden from the problem formulation. In practice, problems are often fairly easy to solve if the network structure is represented in the problem formulation. We show that by representing OneSwitch dataplane as a weighted bipartite edge coloring problem [12], the graph coloring can be done using no more than m colors, where m is the number of core switches, in polynomial time.

The OneSwitch dataplane can be transformed into a weighted bipartite edge coloring problem by representing each edge switch with two vertices's, one representing the origin and the other

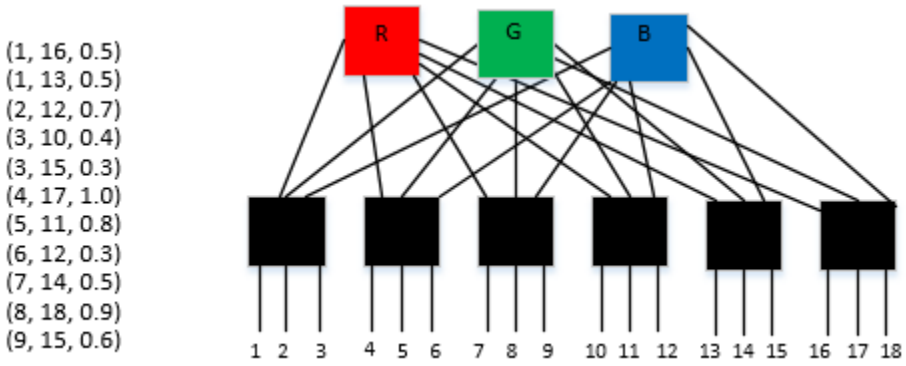

Figure 5.3: Simultaneous Flow Routing.

representing the destination. An edge (arc) is added for each flow that needs to be routed between the origin and destination. Each flow is assigned a weight between 0 and 1 that represents the bandwidth used by each flow. Edges incident to the same edge switch are allowed to have the same color, so long as the total weight of all flows on an edge do not exceed it's capacity. The colors assigned to the edge correspond to the core switches used to carry the flows and the constraint on the colors corresponds to the constraint that the total weight of flows from the same origin (or destination) switch can pass through the same Core switch.

Figure 5.3 shows an example of a set of flows that need to be routed over a OneSwitch data plane with with 3 Core switches and 6 edge Switches. Each flow is represented by 3 fields, (Source, Destination, Weight). The left part of Figure 5.4 shows the corresponding weighted graph for the set of flows. The weighted graph coloring problem can be converted to an ordinary graph coloring problem by splitting each of the vertices's and associating different subsets of the edges incident to the vertex with different sub-vertices's. In particular, the

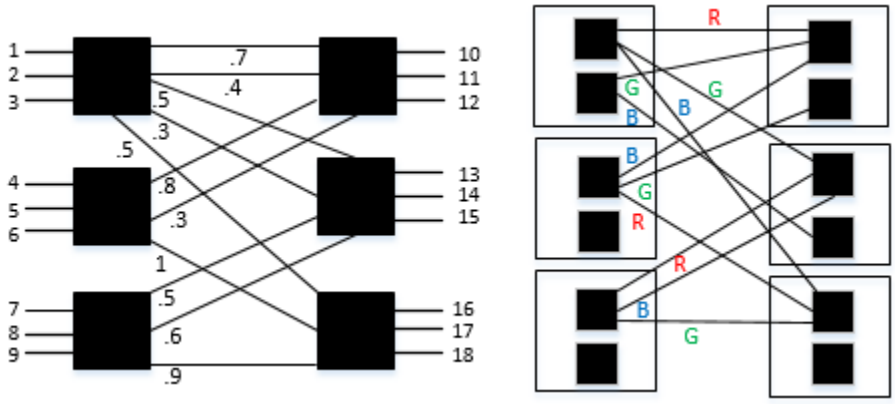

Figure 5.4: Weighted graph coloring in OneSwitch.

m heaviest edges are all assigned to the same sub-vertex, the next m edges are assigned to another sub-vertex, and so forth. This is illustrated in the right part of the Figure 5.4. When this splitting procedure has been applied to all vertices's the resulting graph has at most m edges incident to each vertex and so can be colored in the ordinary way (only one edge of each color), hence the graph coloring can be done using no more than m colors.

The time to find the minimal edge coloring in a bipartite graph is $O(E \log V)$ [5], where E represents Edges and V represents Vertices. In OneSwitch $E = m$ and $V = 2m$, therefore the complexity of the edge coloring method in OneSwitch is approximately $O(m \log m)$, hence simultaneous routing in OneSwitch network can be done in polynomial time.

5.3 Optimal routing without flow collisions

In this section we provide an example of how a general Integer Programming solver can be used in OneSwitch network to optimally route flows without flow collisions. We simulate a

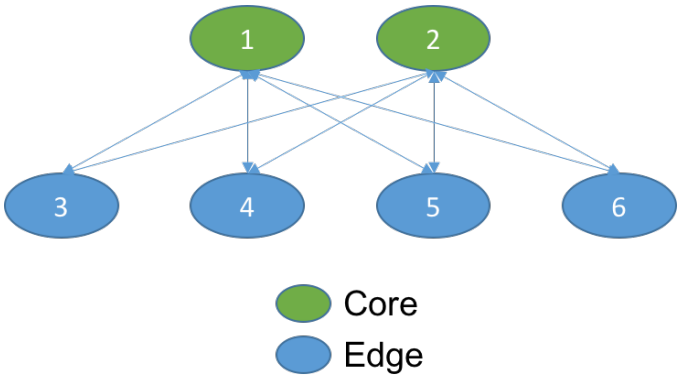

Figure 5.5: OneSwitch dataplane represented as a directed graph.

small OneSwitch network that has a dataplane that consists of 2 core switches connected to 4 edge switches. All connections have the same capacity of 1Gbps. The OneSwitch network is mapped to a directed graph that has two core nodes 1 and 2, and four edge nodes $3, 4, 5$, and 6 as shown in Figure 5.5. There are a total of 16 links, represented by x_{ij}. The term x_{ij}^k represents the decision variable of the k^{th} flow on link x_{ij}. $x_{ij}^k = 1$ if the k^{th} flow is routed over link x_{ij} and $x_{ij}^k = 0$ if the k^{th} flow is not routed over link x_{ij}. In the general case, there are as many paths as there are core switches for the OneSwitch network. In this specific topology, each flow has two possible paths to get from origin edge node o to destination edge node d.

We use Matlab's **intlinprog** [102] mixed integer programming solver to optimally assign flows to paths without flow collisions. The **intlinprog** solver finds the minimum of a problem specified by the following objective function and constraints.

$$\min_{x} f^T x$$

subject to

$$\begin{cases} x(intcon)\text{are integers} \\ Ax \leq b \\ A_{eq}x \leq b_{eq} \\ lb \leq x \leq ub \end{cases} \tag{5.5}$$

We describe how the notation in equation 5.5 maps to the optimization equations given in 5.1, 5.2, 5.3, and 5.4.

The variables used in the **intlinprog** function are $f^T x$, A, b, Aeq, beq, lb, and ub. We use nl and nf to represent number of links and number of flows respectively. $f^T x$ is the objective function to be minimized. Vector x is the output result of the optimization and vector f is an input variable to be programmed. In terms of implementation, the dimensionality of vector x is $nf \times nl$. Thus, x is laid out in memory as follows:

$$\mathbf{x} = \begin{bmatrix} x_{13}^1 \\ x_{14}^1 \\ \vdots \\ x_{42}^1 \\ \vdots \\ \vdots \\ x_{13}^K \\ x_{14}^K \\ \vdots \\ x_{42}^K \end{bmatrix} \tag{5.6}$$

Equation 5.6 shows that there are 16 (Total number of links in this example) x_{ij}^k possible decision variables represented by x_{ij} for the k^{th} flow, with a total of K flows. The f vector, which is of the same dimensionality as the x vector is programmed as follows: f_{ij}^k is the rate of the k^{th} flow if link ij is a possible link which flow k can take to go from origin node i to destination node j. Here, ij is not a literal index into the vector f, but rather a logical location given by equation (5.6).

$Ax \leq b$ describes the link capacity constraint given by equation 5.2. A is a matrix of dimension $nl^2 \times nf$, and the structure of this matrix is shown in Figure 5.6. The A matrix is programmed as follows: A_{ij}^k is the rate of the k^{th} flow if link ij is a possible link which flow k can take to go from origin node i to destination node j, entered into row ij of A. b

is a column vector of dimension nl, and just represents the maximum link capacity. Under the assumptions of our problem statement, each element in b is set to 1Gbps.

$$A = \begin{bmatrix} x_{13}^1\, x_{13}^2 \ldots x_{13}^K & x_{14}^1\, x_{14}^2 \ldots x_{15}^K & \ldots & x_{42}^1\, x_{42}^2 \ldots x_{42}^K \\ x_{13}^1\, x_{13}^2 \ldots x_{13}^K & x_{14}^1\, x_{14}^2 \ldots x_{15}^K & \ldots & x_{42}^1\, x_{42}^2 \ldots x_{42}^K \\ & \vdots & & \\ x_{13}^1\, x_{13}^2 \ldots x_{13}^K & x_{14}^1\, x_{14}^2 \ldots x_{15}^K & \ldots & x_{42}^1\, x_{42}^2 \ldots x_{42}^K \end{bmatrix}$$

Figure 5.6: The structure of the A matrix.

$A_{eq}x \leq b_{eq}$ describes the flow conservation constraints given in equation 5.2. The A_{eq} is a matrix of dimension $(4 * nf) \times (nl * nf)$, and b_{eq} is a column vector of size $4 * nf$. The flow conservation constraints encode which nodes are source nodes (denoted by +1), which nodes are sink nodes (denoted by -1), and which nodes are transient nodes (denoted by 0). In order to encode this using **intlinprog**, we encode the A_{eq} and b_{eq} matrix and vector, respectively, as follows: For each flow, all possible active links that originate from the source node or end at the destination node are set to 1 in both A_{eq} and b_{eq}. The input link to the transient node is set to -1 and the output link from the transient node is set to 1, and for

this case, the corresponding b_{eq} is set to 0. lb and ub map directly to the binary constraints given by equation 5.4.

Both lb and ub are column vectors of size $nl \times nf$. All the elements in lb are set to zero, to specify that the minimum integer value which is a feasible solution is 0, and all the elements of ub are set to one, to specify that the maximum integer value which is a feasible solution is 1. Setting lb and ub in this manner forces **intlinprog** to produce a binary solution, either 0 or 1.

Simulating routing without flow collisions

We continue our analysis by simulating a number of scenarios. For each scenario we find the optimal flow assignment using **intlinprog** solver and graph the link utilization for the network architecture given in Figure 5.5 under the same parametric classes discussed in section 4.6.

Each edge switch has two 1 Gbps links, therefore the total amount of bandwidth for all flows allowed from any edge switch to any other edge switch is defined to be 2 Gbps. Under this assumption, we test with a total flow volume of 1500 Mbps and 500 Mbps from each edge to edge switch. This simulates the network under a relatively high load and light load respectively.

Figures 5.7-5.17 show the bidirectional link utilization of all links between core switches (C1 and C2) and edge switches (E1, E2, E3, and E4), in OneSwitch using a Step (1), Shuffle,

and Random traffic pattern with 10, 100, 1000 flows, and a uniform, predominately mice, and predominately elephant flow size distributions.

In all of the scenarios we simulated we observe that when there are a total of 10 flows, there is a more even utilization of core 1 and core 2 switches. For 100 and 1000 flows, the utilization of core 1 and core 2 is not as uniform as the the scenario where there were 10 flows. Additionally, the core utilization of core 1 and 2 for 100 and 1000 flows is very similar for both the predominately mice and elephant cases. This is due to the fact that there are only 16 links available for a logarithmically increasing number of flows.

We also observe that the ratio of maximum link utilization to minimum utilization in the predominately mice case is higher than the ratio for the predominately elephant case. This is due to the variance in flow size distribution between the predominantly mice and predominantly elephant flows.

In all the scenarios with predominately mice flows we observe a better distribution of link utilization across all links. This is due to the fact that when we have more small flows, we have the flexibility of distributing them more evenly across all the links. We lose this flexibility for larger flows due to the indivisibility constraint of the flows. The only way to overcome this is to increase the capacity of the links, such that the flow size is small relative to the link capacity.

Figure 5.10, 5.14, and 5.18 shows the CPU time required to solve the optimization problems. It can be seen that the in all of the scenario's the amount of CPU time required to solve

the optimization problem increased linearly as the number of flows increased. Additionally, the run time did not change as the flow volume changed. This is to be expected, as the run time is a function of the matrix size, not the values in the matrix.

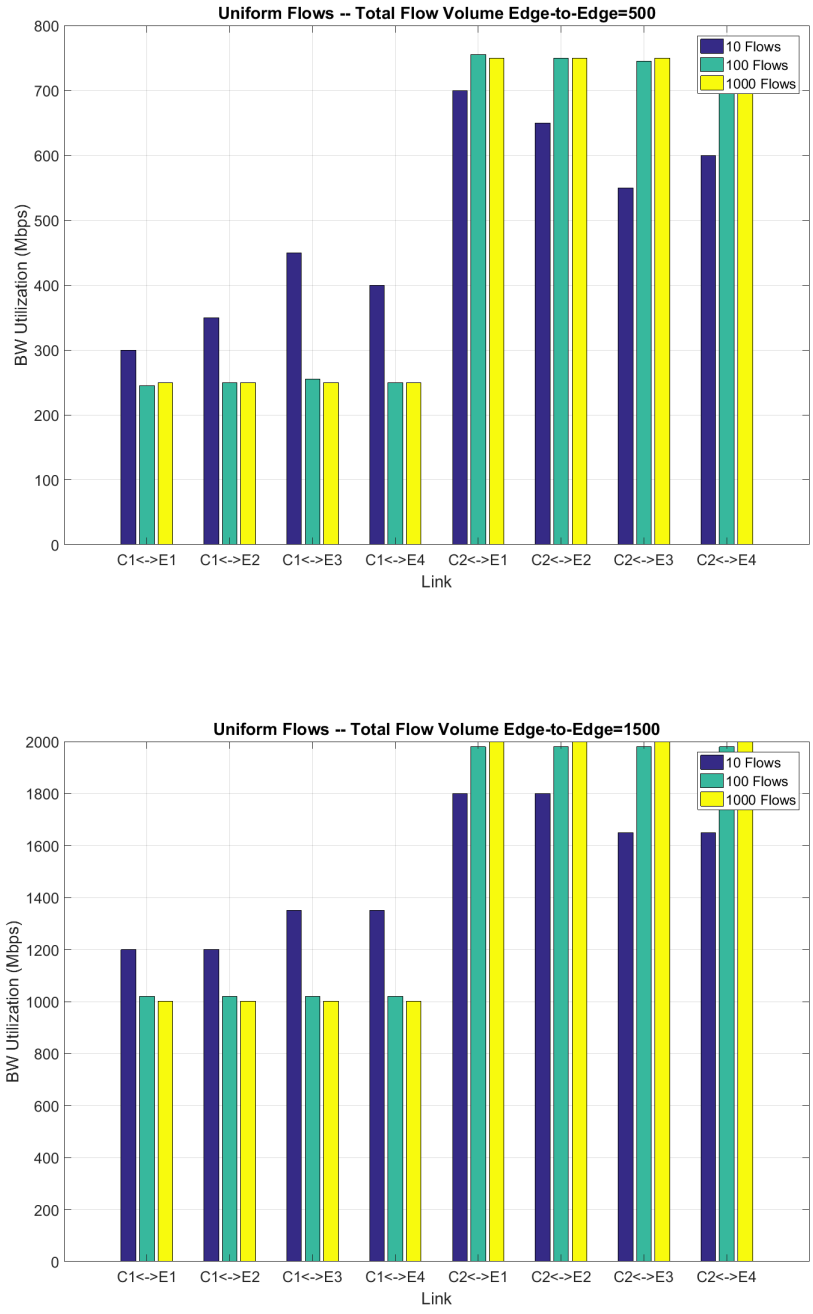

Figure 5.7: Link utilization for uniform flows and step traffic pattern.

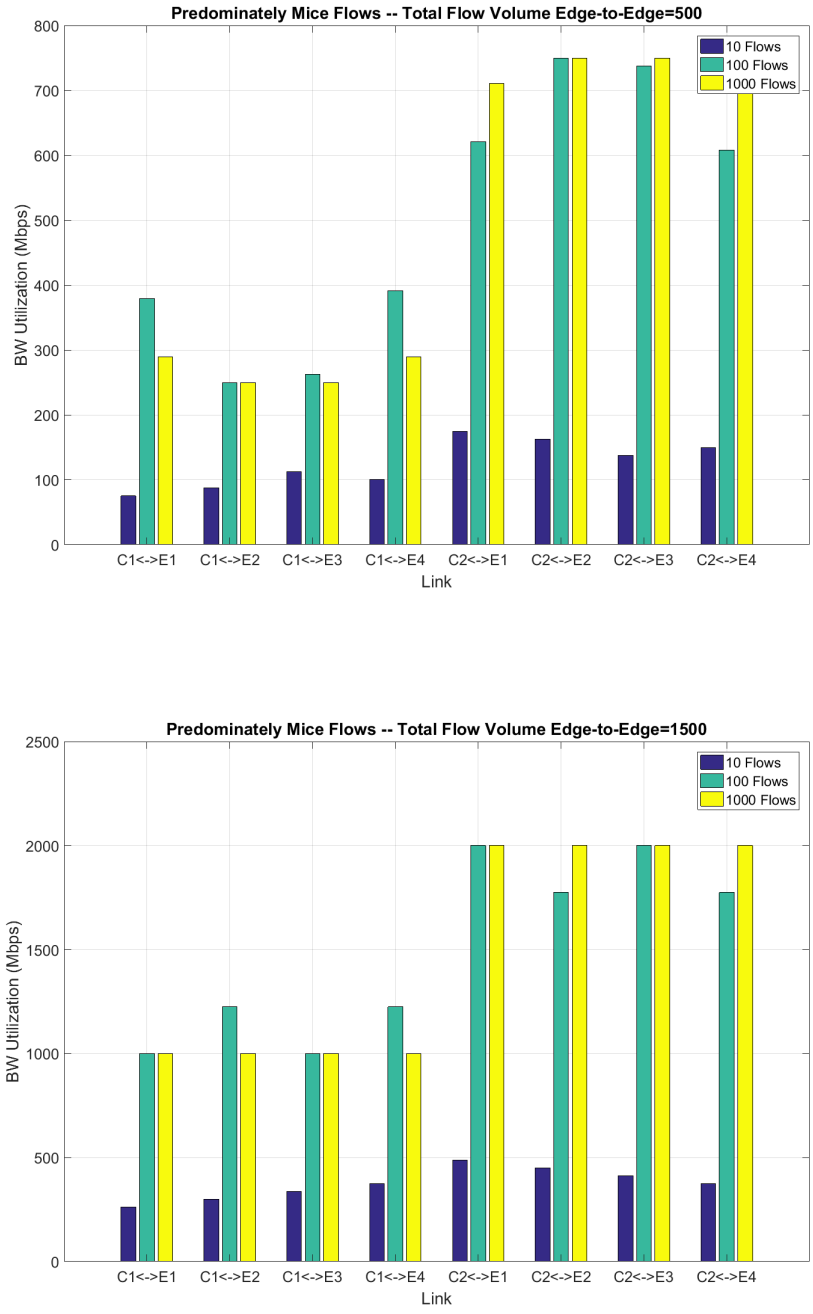

Figure 5.8: Link utilization for predominately mice flows and step traffic pattern.

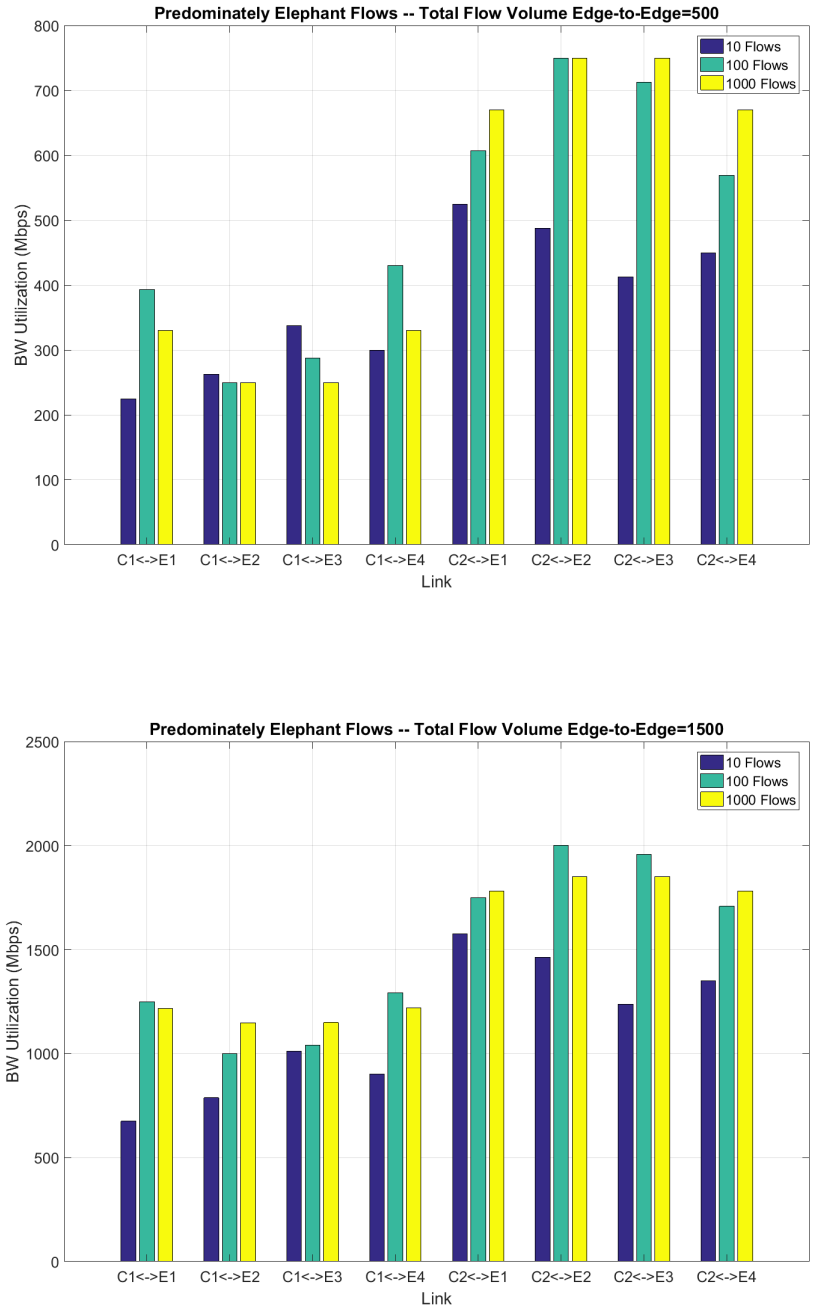

Figure 5.9: Link utilization for predominately elephant flows and step traffic pattern.

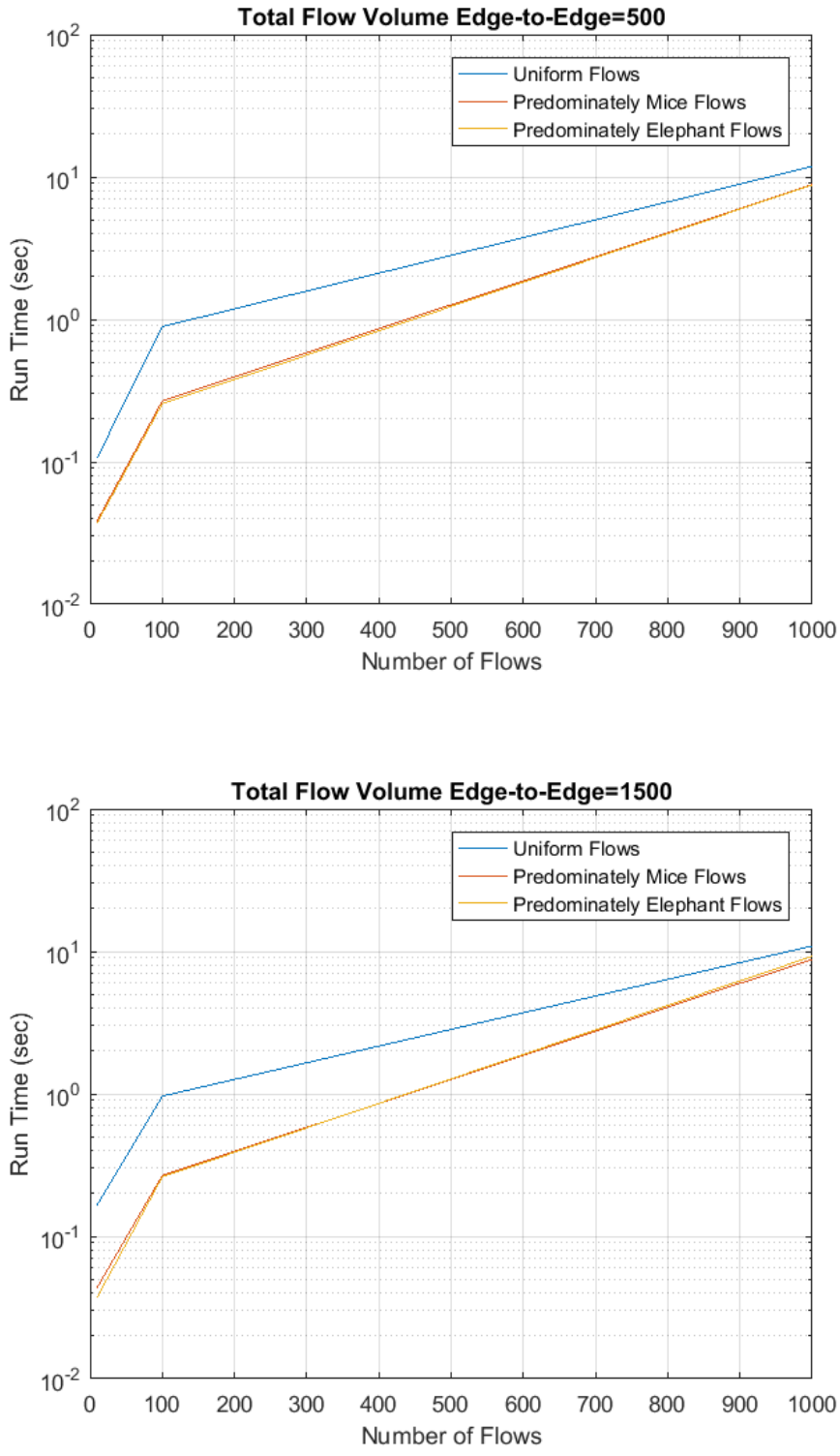

Figure 5.10: Runtime vs Number of flows for different flow types and step traffic pattern.

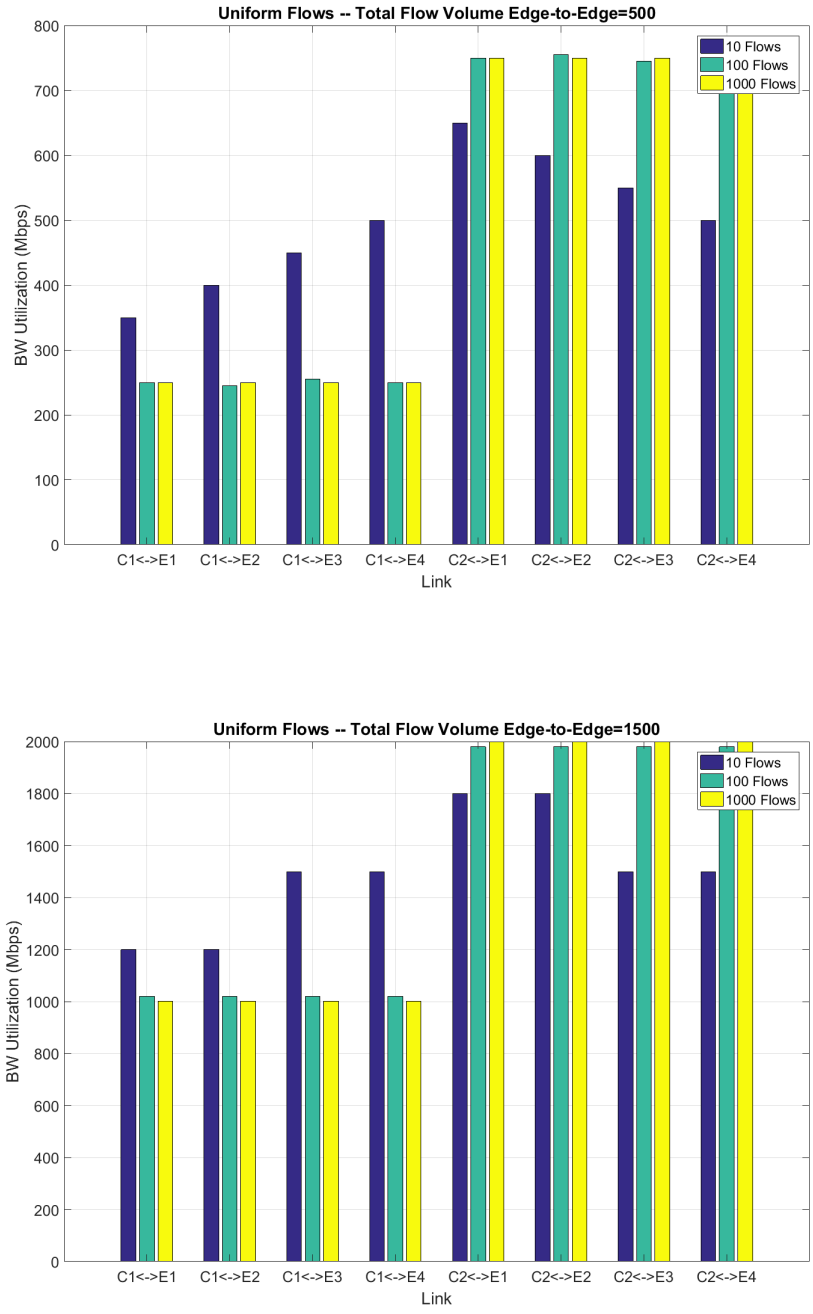

Figure 5.11: Link utilization for uniform flows and shuffle traffic pattern.

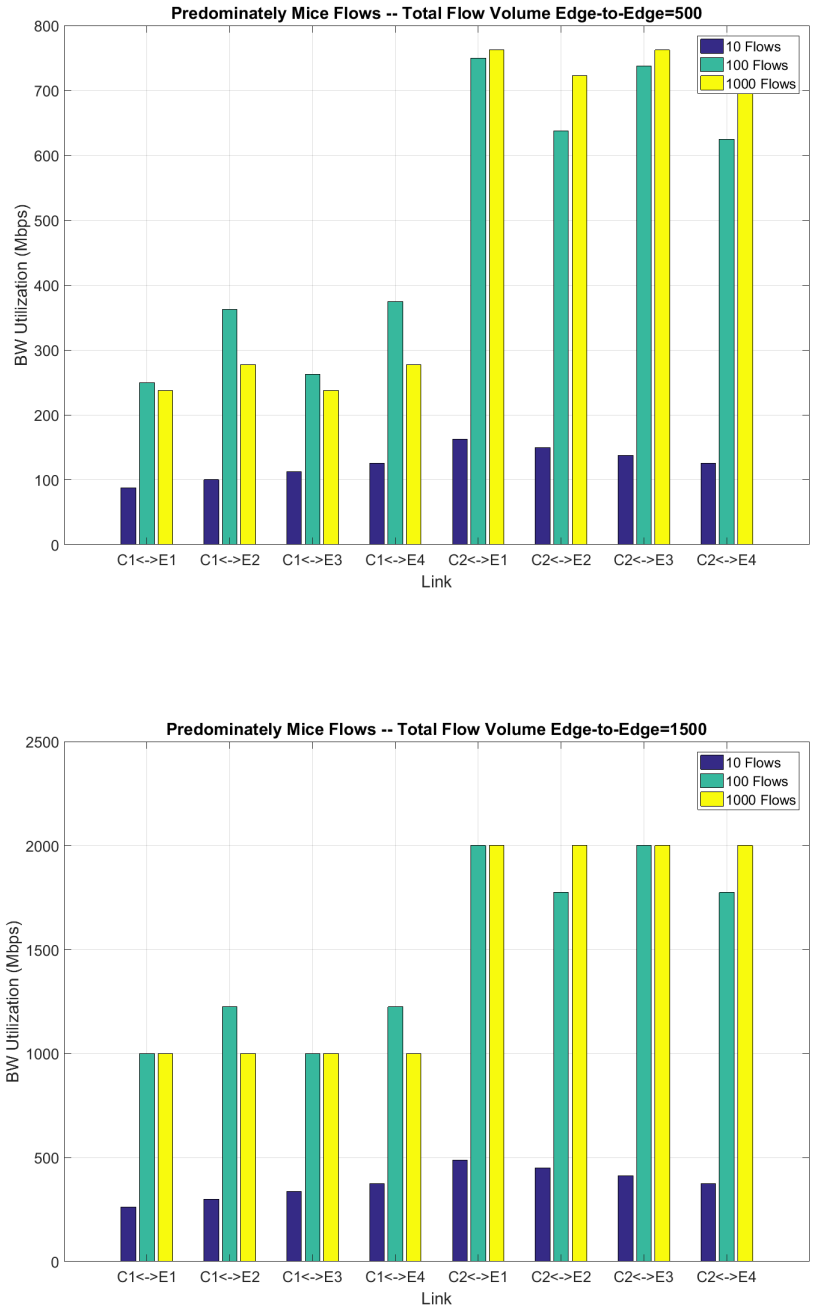

Figure 5.12: Link utilization for predominately mice flows and shuffle traffic pattern.

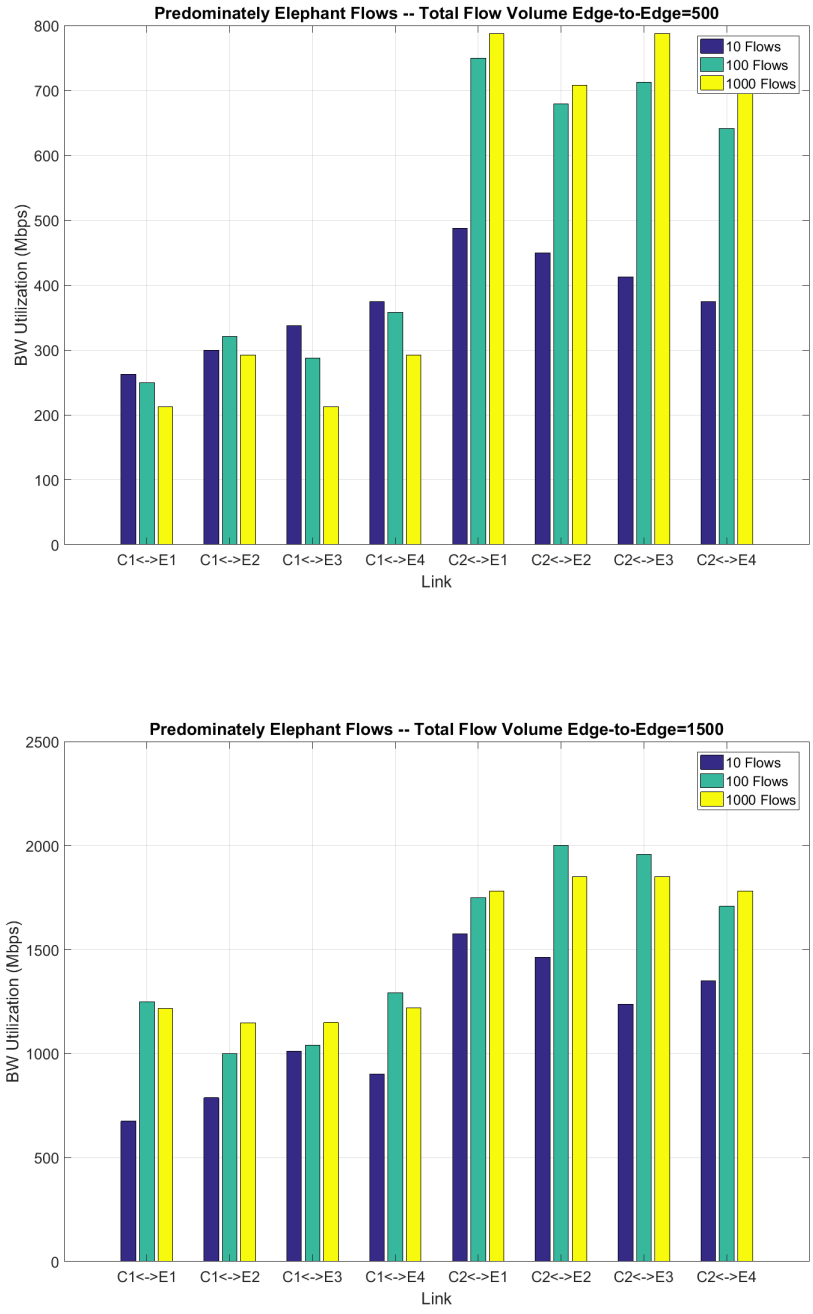

Figure 5.13: Link utilization for predominately elephant flows and shuffle traffic pattern.

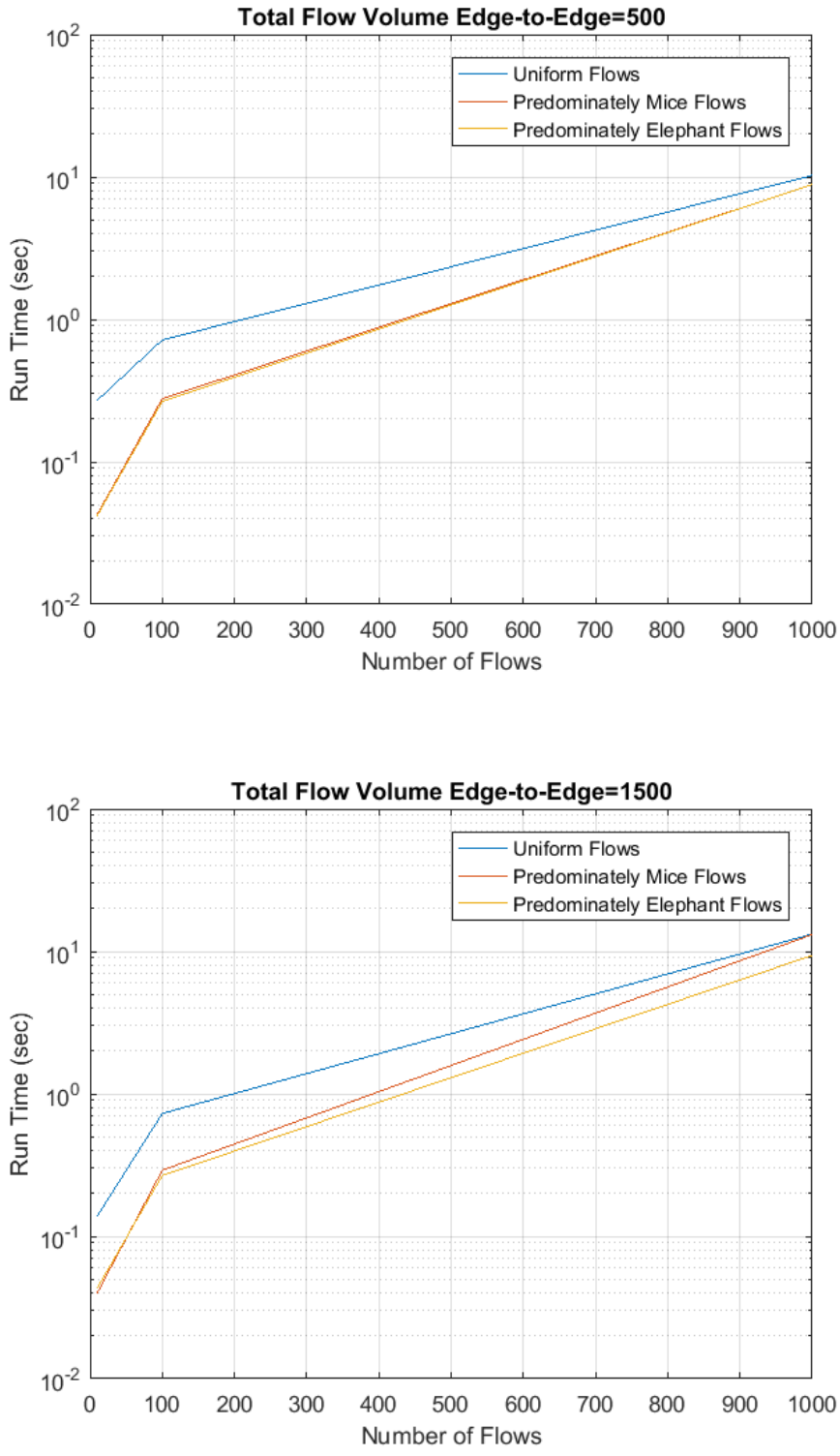

Figure 5.14: Runtime vs number of flows for different flow types and shuffle traffic pattern.

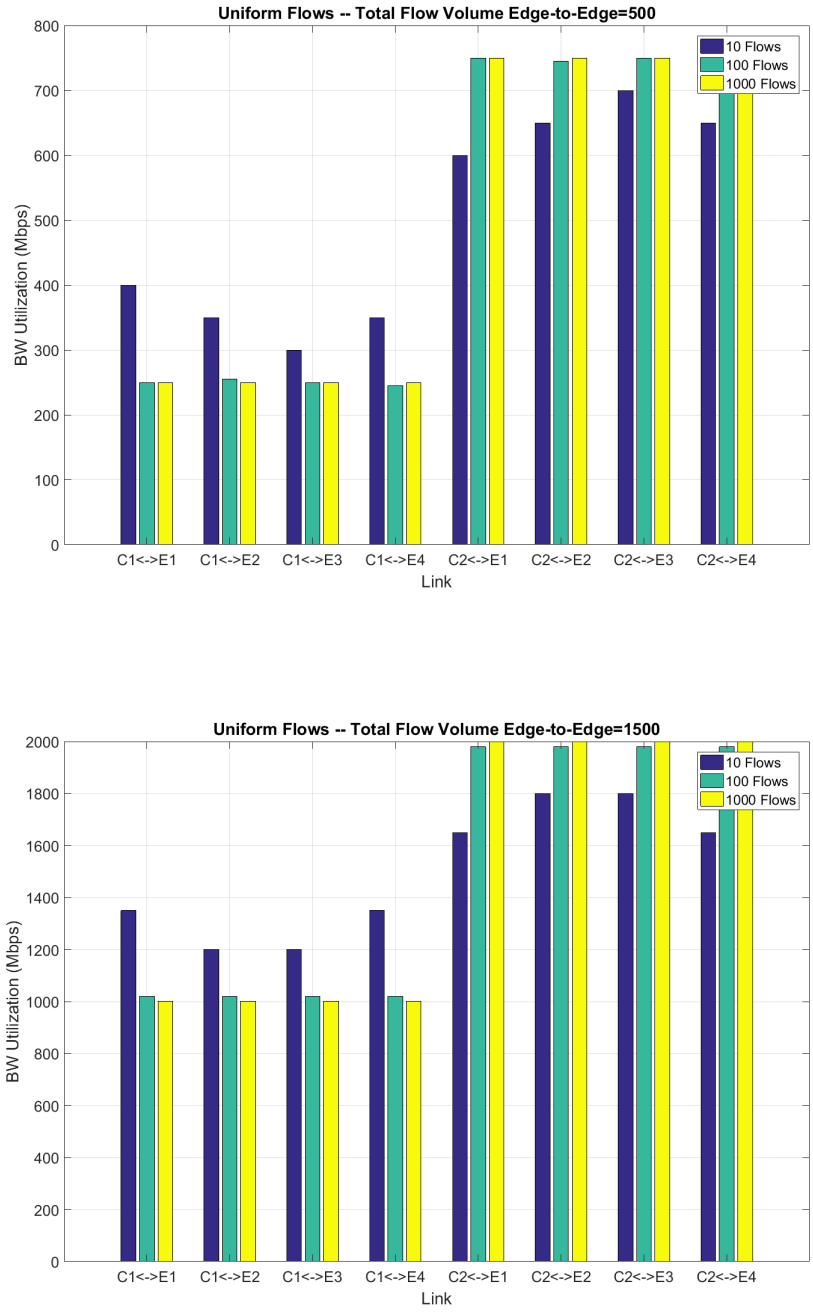

Figure 5.15: Link utilization for uniform flows and random traffic pattern.

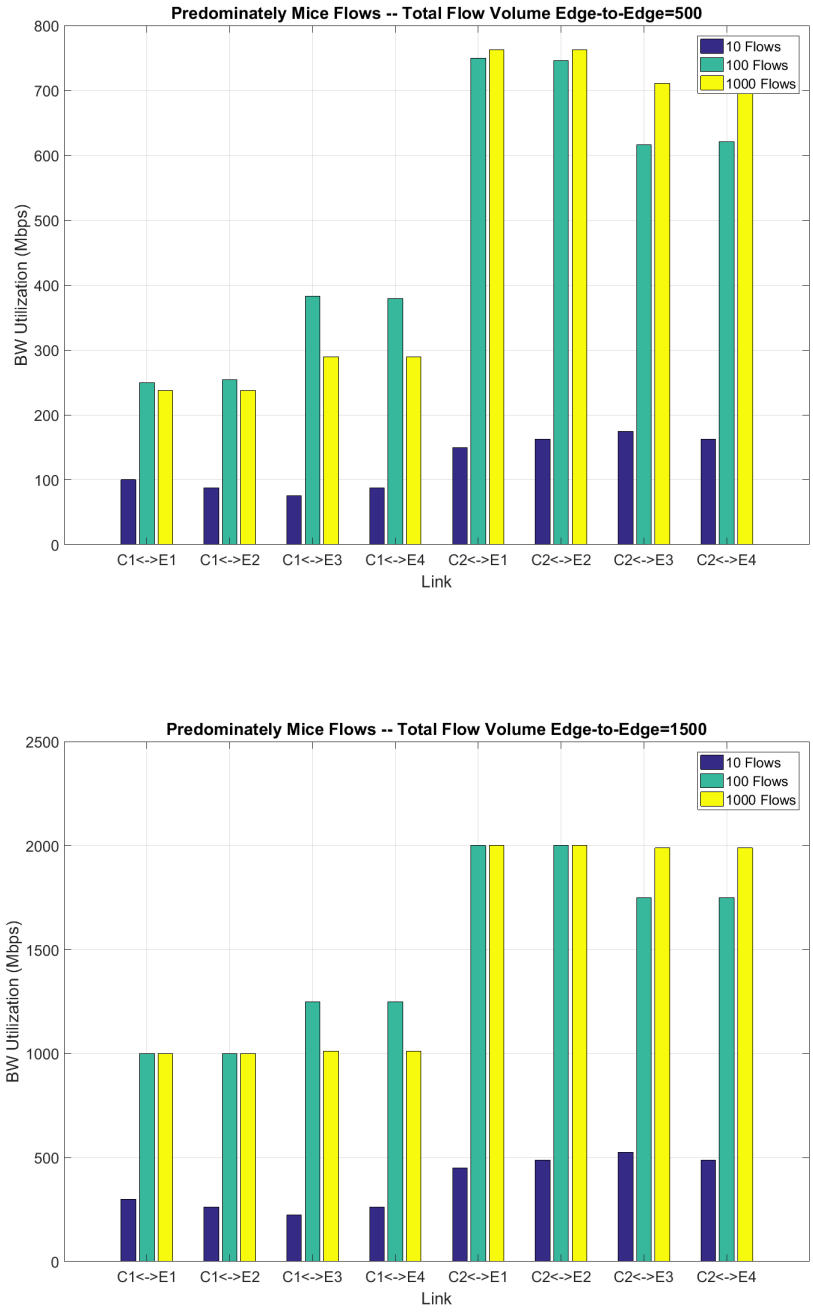

Figure 5.16: Link utilization for predominately mice flows and random traffic pattern.

Figure 5.17: Link utilization for predominately elephant flows and random traffic pattern.

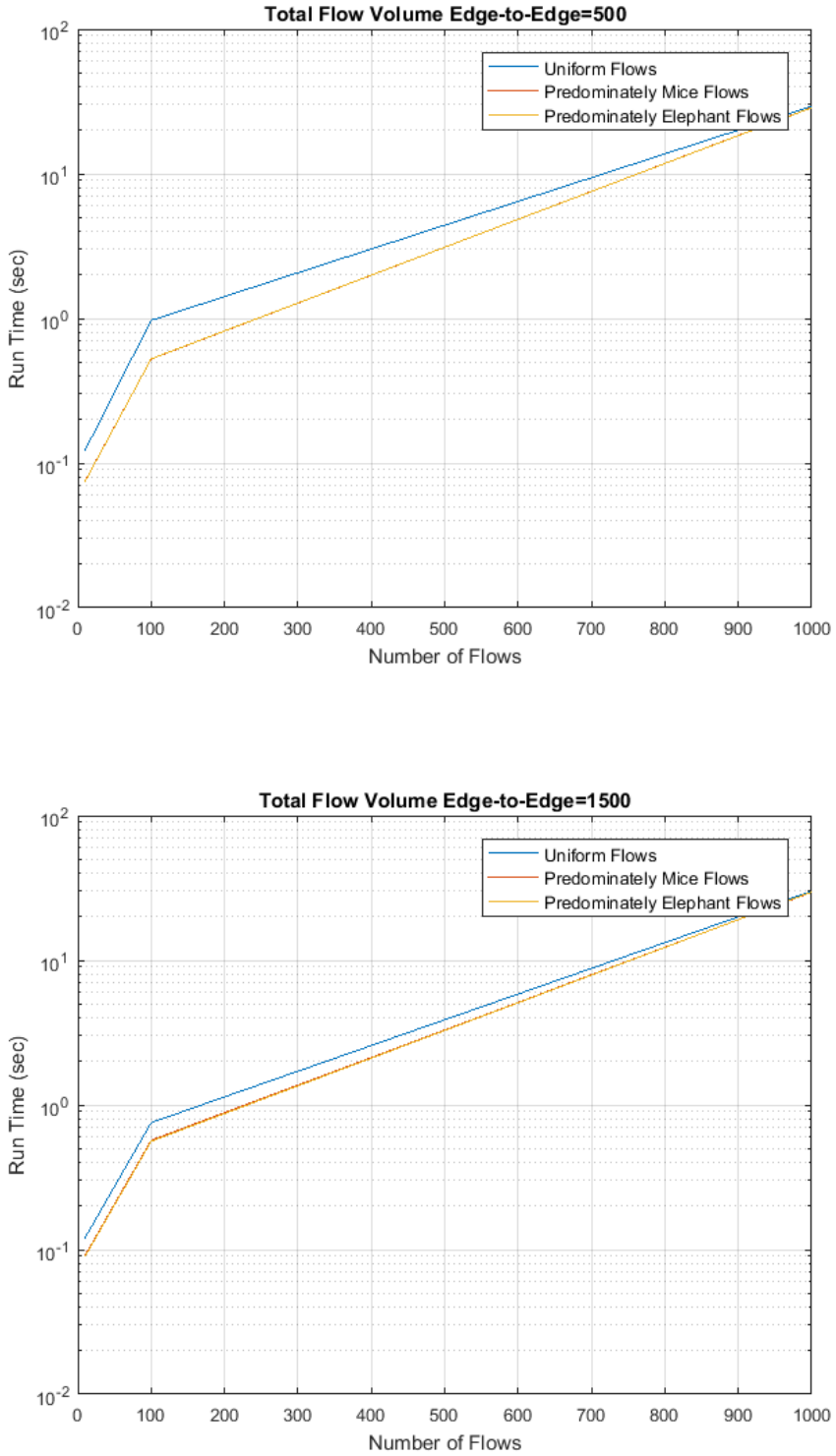

Figure 5.18: Runtime vs number of flows for different flow types and random traffic pattern.

It should be noted that in all of the scenarios we tested, a feasible solution was found. This is due to the fact that OneSwitch uses a balanced non-blocking Folded Clos architecture: The total downlink bandwidth for each edge switch is equal to to the total uplink bandwidth. If an unbalanced Folded Clos architecture was used, feasible solutions may not be found if the bandwidth of transmitted flows exceeded the total capacity of the architecture. In this scenario, a non-optimal solution must be found.

5.4 Optimal routing with minimal flow collisions

5.4.1 Optimized Flow Re-routing (OFR)

In the previous section we used the mixed integer programming solver **intlinprog** to optimally assign flows to paths without flow collisions. In a real network it may not be practical to re-route all existing flows every time a new flow needs to be routed. A more practical solution would be to periodically probe the network and optimally re-route just elephant flows. This approach has a significant impact because it reduces the number of flows that need to be re-routed, which in turn dramatically reduces the number of decision variables and constraints that need to be optimized. In the case of using the **intlinprog** solver, this leads to smaller matrices, allowing for an optimal solution to be reached in a reasonable time.

We present Optimized Flow Re-routing (OFR): a modified version of the Binary Multi-

commodity Flow problem formulation discussed in the previous section. OFR formulation optimally re-routes just elephant flows at periodic monitoring intervals by making slight changes to the original problem formulation as follows:

In the original formulation r^k in equation 5.1 represented the rate of the $k'th$ flow in set K. In OFR formulation r^e represents the rate of only elephant flows in set K at every monitoring interval. Also in equation 5.1, c_{ij}^k was set to 1, because all of the links had the same available capacity, u_{ij}, and hence the cost to route a flow over any link ij was the same.

In OFR formulation, because we only re-route elephant flows at every monitoring interval, we have to take the current state of the network into consideration. To do so we introduce a new variable z_{ij} to represent the current utilization on link ij and z_{ij}^e to represent the cost to route an elephant flow e over link ij. The variable z_{ij}^e replaces c_{ij}^k in equation 5.1. By doing this we let the current link utilization be used as a cost value for each link ij. The higher the link utilization the more costly it is to re-route an elephant flow over it. The link capacity constraints in equation 5.3 will also change according to the current link utilization at each periodic interval, and is represented by $u_{ij} - z_{ij}$.

We now can express the objective function in OFR problem formulation as follows:

$$min \sum_{k \in K} \sum_{ij \in \mathcal{A}} z_{ij}^e r^e x_{ij}^e \tag{5.7}$$

.

The flow rate constraint is expressed as:

$$\sum_{j:ij\in\mathcal{A}} x_{ij}^{e} - \sum_{j:ji\in\mathcal{A}} x_{ji}^{e} = \begin{cases} 1 \; i = o^{e} \\ -1 \; i = d^{e} \\ 0 \; i \neq o^{e},\, i \neq d^{e} \end{cases} \tag{5.8}$$

The link capacity constraint is now expressed as:

$$\sum_{e\in K} r^{e} x_{ij}^{e} \leq u_{ij} - z_{ij}, \forall \; ij \in \mathcal{A} \tag{5.9}$$

The indivisibility of flows is expressed as:

$$x_{ij}^{e} \in \{0,1\}, \; \forall e \in K, \; \forall ij \in \mathcal{A} \tag{5.10}$$

Although the above formulation is guaranteed to find a feasible solution, the solution maybe not be the most optimal. This is because the optimization does not take into consideration all flows in the network. By only re-rerouting elephant flows OFR reduces the optimization complexity from $O(AF)$ to $O(EF)$, where AF and EF represent all flows and elephant flows respectively and $EF \ll AF$.

OFR's heuristic approach doesn't eliminate the possibility of flow collisions, but rather minimizes them. By using the OFR problem formulation we can easily use any mixed integer programming solver to periodically re-route elephant flows in order to achieve better link utilization and at the same time minimize flow collisions. The frequency of the periodic monitoring interval and how elephant flows are identified are important performance factors

that are not mentioned here. A detailed discussion of these parameters is provided in section 5.4.4.

5.4.2 FlowFit

In the previous section we discussed how OFR can be used to minimize the number of flows that need to be optimized by just re-routing elephant flows, however that could still mean changing many of the existing routing's, which would be burdensome. In this section we look at further optimizing the re-routing of elephant flows by only re-routing a subset of elephant flows. The goal in such a case is to achieve an optimal solution with as small changes as possible to the existing routing's.

We propose FlowFit, a practical flow optimization algorithm that periodically monitors the state of the network and targets the re-routing of only the largest elephant flows on a congested link, to a non congested link, one by one until a link is no longer congested. FlowFit insures a quick and effective way of relieving congestion, increasing bisection bandwidth, and reducing flow completion times, by only re-routing a small subset of elephant flows.

To help understand how the FlowFit algorithm works we identify the following terms.

- Flow Bandwidth (FBW) is the current bandwidth of an individual flow.
- Link Load (LL) is the current total bandwidth used by all flows on the link.
- Link Capacity (LC) is the maximum bandwidth that a link can support.

- Over-loaded link (OLL) is a link that has a link load $\geq$ half the links capacity.
- Under-loaded link (ULL) is a link that has a link load $<$ half the links capacity.
- The Monitoring Interval (MI) is 5 seconds and is the time that elapses before the Flow Fit algorithm is triggered.
- Elephant Flow (EF) is a flow that has a sustained bandwidth of greater than 0.05 percent of the links capacity for more than the monitoring interval.

For example if we had a 1Gbps link that was carrying 10 flows and each flow had a sustained bandwidth of 60Mps for 5 seconds or more, then all these flows would be classified as elephant flows, because they are over the 50Mbps threshold for elephant flows (0.05 X 1Gbps). The Flow Bandwidth would be 60Mps, the Link Load would be 600 Mbps (10 Flows X 60 Mbps) and the Link Capacity would be 1Gbps.

FlowFit algorithm can be enabled in a network running ECMP or any other routing method. When enabled, FlowFit runs at the end of each monitoring interval. The bandwidth of each link is measured, and links are categorized into two groups, Over-Loaded Links and Under-Loaded Links. If an Over-Loaded Link is found, the largest elephant Flow on the link will be re-routed to the most Under-loaded Link. If after moving the first largest elephant flow, the link is still an Over-Loaded Link, the next largest elephant flow is moved to the next most Under-Loaded Link. This is continued until the Overloaded-link becomes an Under-Loaded Link. If during the monitoring interval, multiple Over-Loaded links are found, then the same procedure is repeated for each Over-Loaded Link, starting with most Over-Loaded Link, then

Algorithm 2 FlowFit

1: **procedure** At Time T=t_0, Poll links for FBW & LL statistics

2: $FBW_{t0} = FlowBandwidth\{link : flows\}, LL_{t0} = LinkLoad\{link : load\}$

3: At Time T=t_m poll all links for FBW & LL statistics

4: $FBW_{tm} = FlowBandwidth\{link : flows\}, LL_{tm} = linkload\{link : load\}$

5: $FBW_{tm-t0} = FBW_{tm} - FBW_{t0}$

6: **for all** $link$ in FBW_{tm-t0} **do**

7: **if** $flows > 0.05$ LC **then**

8: $flows$ is EF, $EF = ElephantFlow\{link : flows\}$

9: Sort all $flows$ in EF **descending**

10: **for all** $link$ in LL_{tm} **do**

11: **if** $load \geq 0.5$ Link Capacity **then**

12: $link$ is OLL, $OLL = OverLoaded\{link\}$

13: **else**

14: $link$ is ULL, $ULL = UnderLoaded\{link\}$

15: Sort OLL **descending** and ULL **ascending**

16: **for all** $link$ in OLL **do**

17: **for all** EF in $link$ **do**

18: **if** $load$ in $link > 0.5$ LC **then**

19: Move EF to $link$ in ULL

20: $load = load - EF$

21: Repeat every T=t_m

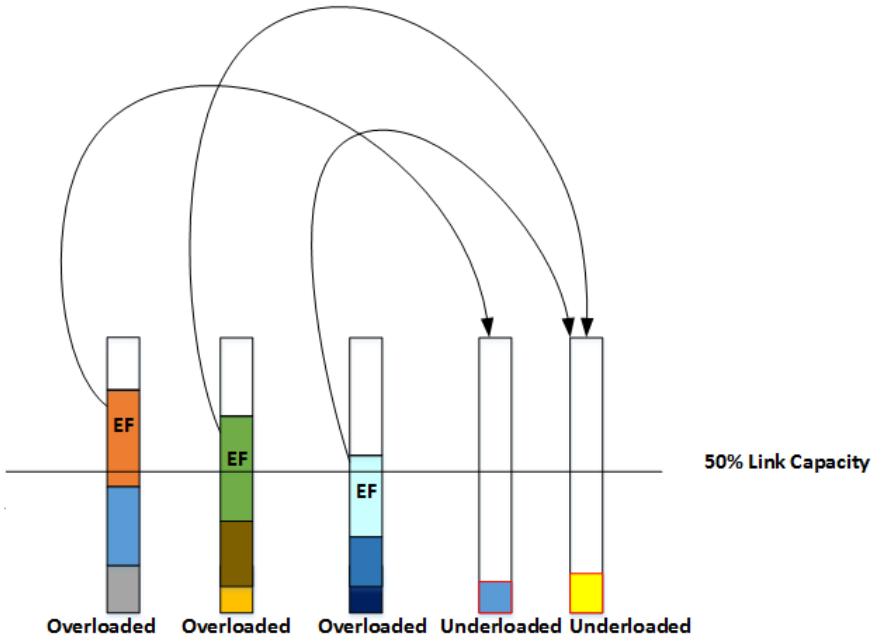

Figure 5.19: Elephant flow re-routing in FlowFit.

the next and so on. Figure 5.19, shows how elephant flows are re-routed in FlowFit. The pseudo-code is also shown in Algorithm 2.

5.4.3 Simulating routing with minimal flow collisions

We simulate routing with minimal flow collisions by modeling the OneSwitch dataplane as a network graph with capacitated edges, as described in section 5.2. In order to simulate large data centers we model TCP flows in steady state only at constant rates. By eliminating TCP slow start and AIMD phases and using fixed rates the simulator can ignore per-packet behavior, which in turn increases scalability. We use the same 3 parametric classes discussed in sections 4.6 and 5.3 as inputs to the simulator.

Flows are created and purged using a Poisson distribution and edge capacities are updated accordingly. The flow rates are constrained by the available bandwidth on host links. The flows are routed to their destinations by randomly assigning them to an edge connecting to one of the core switches, simulating EMCP routing. A flow collision is recorded anytime a flow is assigned to a congested edge causing it to exceed its capacity.

At every monitoring interval we run FlowFit and OFR to optimize the assignment of flows. If the bandwidth of an edge drops below its maximum capacity, as a result of flow optimization, a flow collision reduction is recorded. We simulate a large data center with $K = 72$ switches and 1152 hosts. We run 5 rounds of tests, calculate the average and standard deviation of normalized flow collision for each round, and finally plot the results with 95% confidence interval.

Figures 5.20, 5.21, and 5.22 show the number of flow collisions normalized to the number of total flows for ECMP, OFR, and FlowFit for 10, 100, 1000 flows using step, shuffle, and random traffic patterns with uniform, predominately mice, and predominately elephant flow sizes.

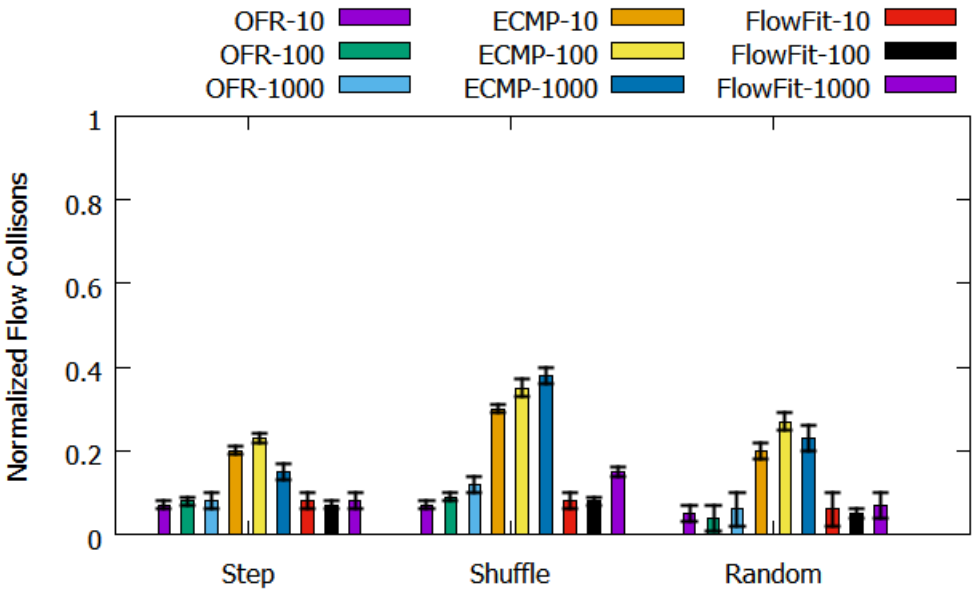

Figure 5.20: Normalized flow collisions-Predominately mice.

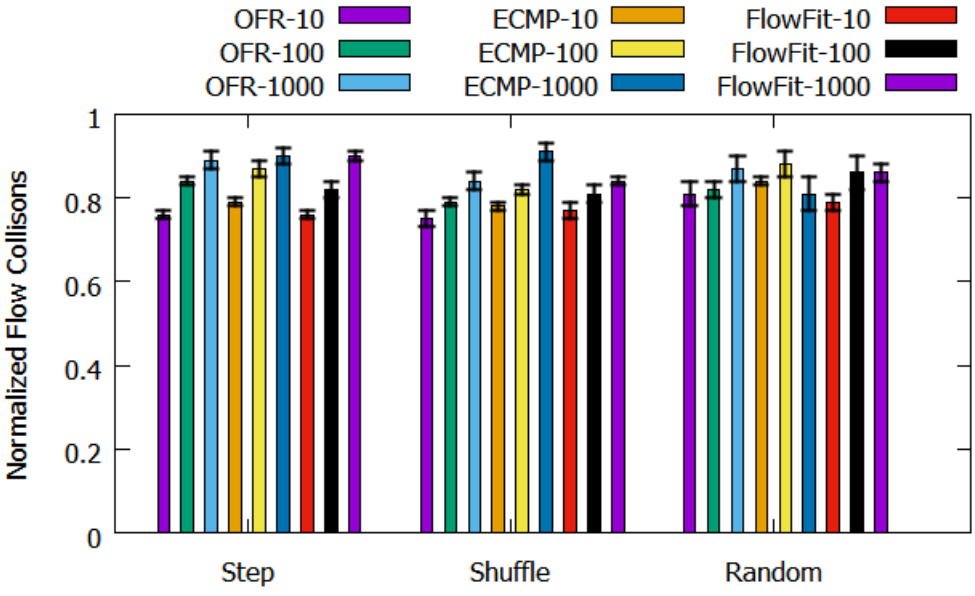

Figure 5.21: Normalized flow collisions-Predominately elephant.

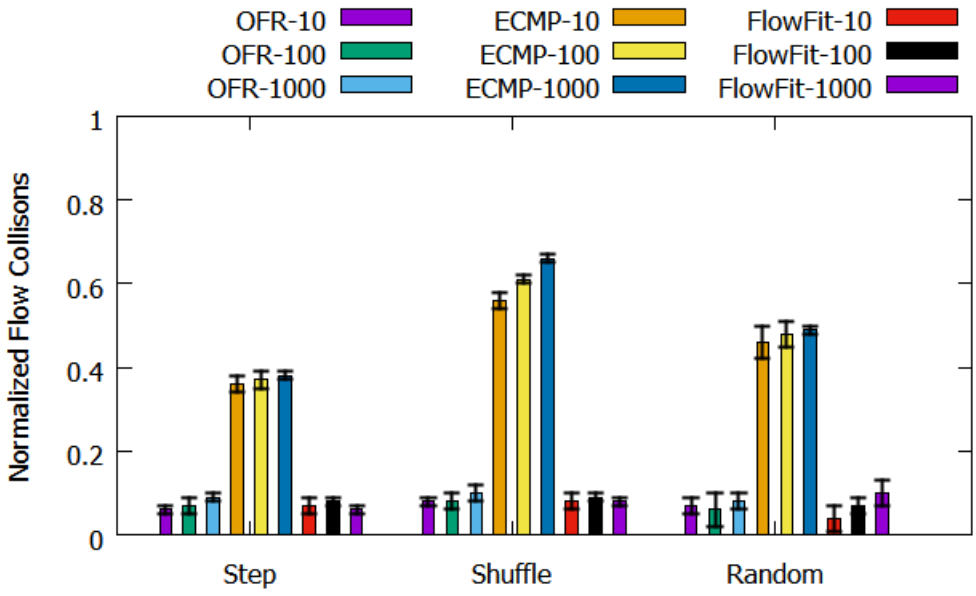

Figure 5.22: Normalized flow collisions-Uniform.

We observe that the scenario with predominately elephant flows has the most flow collisions when using ECMP followed by uniform flows and then predominately mice flows. We observe a reduction in flow collisions when flows are optimized using OFR and FlowFit for all flow sizes.

5.4.4 FlowFit and OFR Design Considerations

Next we discuss some practical considerations and design choices for some of the parameters used for OFR and FlowFit. Both OFR and FlowFit re-route elephant flows at the end of a monitoring interval. This raises two practical considerations, how is an elephant flow detected, and what is the frequency of the monitoring interval.

For how an elephant flow is detected, one obvious method is to collect packets that belong to all flows and then extract their flow statistics. A link at any given time may contain different types of flows. Knowing the sizes of all flows lacks scalability, especially for high speed links [24, 29]. A more practical solution is packet sampling. The simplest method of packet sampling is periodic sampling.

In periodic sampling a flow is measured at certain intervals in order to determine its size. Periodic sampling is not effective for measuring the size of mice flows, because they are short lived. Elephant flows on the other hand are large and long lived making them easier to detect and track. Once a large flow is detected the next step is to identify it as an elephant flow.

A simple method for identifying an elephant flow is by setting a threshold for the flow's size that's sustained for more than the monitoring interval. In OFR and FlowFit the threshold is a percentage of the links capacity. The same monitoring interval that is used to identify elephant flows is also used to trigger the re-routing of elephants. Although elephant flow identification and re-routing can each have their own monitoring interval, we choose to combine them together to minimize the number of times the network is polled, which allows for increased scalability.

As for the frequency, the more frequently the network is monitored the faster elephant flows can be detected. The downside is this can lead to tracking a high number of flows that may not end up being elephant flows. In addition, because the monitoring interval is used to trigger the re-routing of elephant flows, frequent monitoring may cause the network to overreact to unnecessary transient congestion. The monitoring interval can be fine-tuned

depending on the environment. Monitoring times that range from 100's of Microseconds to Seconds have been proposed for detecting elephant flows [93].

FlowFit minimizes the re-routing of elephant flows by only targeting overloaded links. We use a percentage of a links capacity as a threshold to identify an overloaded link, similar to how we identified elephant flows. There is a trade-off on the threshold value. On one hand the higher the threshold the less number of elephant flows that need to be re-routed. On the other hand the network might be too congested if the threshold was too high for the re-routing to be effective.

The re-routing of flows needs to happen as soon as the first signs of congestion are observed, in order to proactively minimize flow collisions. A percentage of half of the links capacity is the minimum threshold that can be used to distinguish between an overloaded and underloaded link and provides the most flexibility for FlowFit to reroute flows on other links. We believe half the capacity is a reasonable early sign of congestion. As the overloaded link threshold increases the network becomes more congested and the re-routing of elephant flows becomes less effective.

The ratio between the overloaded link threshold and elephant flow threshold is important because it determines the maximum number of flows that are rerouted. The ratio is an upper bound on the number of rerouted flows in the worst case scenario, where a links is at fully utilized and all flows are elephants. For example in our implementation of FlowFit we set the overloaded link threshold and elephant flow threshold to 0.5 and 0.05, respectively. The ratio in this case is 10, which means that in a worst case scenario 10 flows will need to be

rerouted to convert an overloaded link to underloaded link.

In summary the performance of OFR and FlowFit rely on the elephant flow threshold and the frequency of the monitoring interval with FlowFit relying on an additional parameter the overloaded link threshold. If some prior knowledge about traffic flows is known then these parameters can be tweaked to achieve the best performance, otherwise these parameters have to be chosen in a way that strikes a balance between performance and scalability.

In our implementation we select half of a links capacity for the overloaded link threshold, 0.05 of the links capacity as the threshold for elephant flows and 5 seconds as the monitoring interval. We observe superior results over ECMP when using these settings with different traffic patterns

Chapter 6

OneSwitch Evaluation

In this chapter we present a detailed evaluation of OneSwitch. We evaluate the scalabilty of both the control and data plane. We also evaluate the Routing Service in OneSwitch by analyzing the performance of SRL, OFR, and FlowFit using Mininet. In addition we discuss the practically of OneSwitch and compare OneSwitch to other schemes used in data center networks.

6.1 Control Plane Scalability

The control plane scalabilty of OneSwitch relies mainly on the POX controller since its used to perform all of the control function in OneSwitch. In this section we evaluate the POX controllers scalabilty using open source bench marking tools. We also analyze the impact of the Location Database on the scalability of the POX controller.

6.1.1 POX Controller

The POX controller in OneSwitch manages all flow decisions in the network, therefore the switches in the OneSwitch architecture can be relatively simple since forwarding decisions are defined by the POX controller, rather than by switch firmware. As the number of switches and flows increases the reliance on a single controller for the entire network might not be feasible.

We analyze the scalability of the POX controller in order to determine if OneSwitch can adequately support a typical production data center network. The POX controller uses the OpenFlow specification. The specification does not specify the number of switches controlled or the proximity of the switch and controller. The reason is that these parameters and others depend on the network design and applications using OpenFlow. A set of parameters that work for one network may not necessarily be suitable for another network.

We investigate 3 main factors that impact the scalability of the POX controller, the number of controllable switches, the POX controller response time, and the bandwidth consumed by control message sent to the POX controller.

Number of controllable switches

The POX controller in OneSwitch is configured in reactive mode. In this mode the POX controller listens to switches passively and configures routes on-demand. After a time out unused entries are flushed out of the switches' flow table.

The reactive behavior allows the POX controller to program switches with flow entries according to the algorithm used in OneSwitch. The reactive mode provides granular control over flows, however this increases the amount of information that needs to be sent to the POX controller raising scalability concerns.

To obtain a baseline for the performance of the POX controller in OneSwitch we use cbench [103]. Cbench emulates an OpenFlow network with a variable number of switches. All switches are OpenFlow enabled, and send new flow requests to the controller. Cbench then records response statistics.

The flow requests do not represent any specific network topology, but are artificially generated in order to provide the illusion that the POX controller is receiving flow requests from a large network. As a result, we obtain the number of OpenFlow messages the controller can support per second.

Cbench supports two operational modes: latency mode and throughput mode. In latency mode, cbench sends one request (i.e, Packet_in) to the controller and waits for the response (i.e, flow_mod) before sending the next request. In throughput mode, cbench tries to send as many requests simultaneously so as to find the maximum controller throughput. More precisely, cbench adapts the request rate to reach a balance, where the number of Packet_in equals the number of flow_mod, which defines the controller's throughput. The pseudo-code for cbench is shown in Algorithm 3.

The cbench tool can be used with the following parameters:

Algorithm 3 cbench

```
1: procedure SIMULATE N SWITCHES (N=16 IS DEFAULT) THEN CREATE AND SEND N
   OPENFLOW SESSIONS TO THE CONTROLLER
2:     if In latency mode (default) then foreach session
3:         Send a Packet_in
4:         Wait for a matching flow_mod to come back
5:         Repeat
6:         Count how many times number 3-5 happens per second
7:     else
8:         In throughtput mode
9:         foreach session and while buffer not full
10:        queue Packet_in's and count flow_mod's as they come back
```

- -c: controller IP
- -p: controller listening port
- -m: Duration per test
- -l: trials per test
- -s: Number of emulated switches
- -M: Number of unique MAC addresses(hosts)per switch
- -t: throughput testing

Figure 6.1 shows an output sample of cbench tool.

Since our goal is to saturate the controller we use cbench in throughput mode. In a normal OpenFlow network when a switch performs a lookup on a destination address and the switch has no such address in its flow table the switch makes a query to the OpenFlow controller. The controller will decide what action would be applied to this flow and programs the switch with an OpenFlow rule. To optimize the memory usage in the switches, OpenFlow rules have an Idle timeout value that sets the removal of flow entries from the flow table. In our experiment we set the Idle timeout value to 10 seconds. A detailed explanation of the impact of the Idle timeout on scalability is presented in the next section.

We evaluate the performance of the POX controller using cbench running on a server that has a dedicated 2.2 GHz Intel core i7 processor and 16 GB of memory. We perform 5 trials

```
cbench: controller benchmarking tool

   running in mode 'throughput'

   connecting to controller at localhost:6633

   faking 16 switches offset 1 :: 3 tests each; 10000 ms per test

   with 10 unique source MACs per switch

   learning destination mac addresses before the test

   starting test with 0 ms delay after features_reply

   ignoring first 1 "warmup" and last 0 "cooldown" loops

   connection delay of 0ms per 1 switch(es)

   debugging info is off

16:53:14.384 16  switches: flows/sec:  18  18  18  18  18  18  18
18  18  18  18  18  18  18  18  18   total = 0.028796 per ms

16:53:24.485 16  switches: flows/sec:  20  20  20  20  20  20  20
20  20  20  20  20  20  20  20  20   total = 0.031999 per ms

16:53:34.590 16  switches: flows/sec:  24  24  24  24  24  24  24
24  24  24  24  24  24  24  24  24   total = 0.038380 per ms

RESULT: 16 switches 2 tests min/max/avg/stdev =
32.00/38.38/35.19/3.19 responses/s
```

Figure 6.1: Cbench sample output.

of tests with varying number of switches sending flow requests to the POX controller and calculate the average controller throughput versus the number of switches.

Figure 6.2 shows how many OpenFlow messages per second the POX controller can process as a function of the number of switches connected to it. The emulation tests were conducted by keeping the server processor and memory state below 95% capacity.

From our tests we can deduce that the POX controller, running on a server with average spec's, reaches its capacity in a network with about 250 switches. We note that 250 switches here represent a highly active network, where switches are producing an extremely large

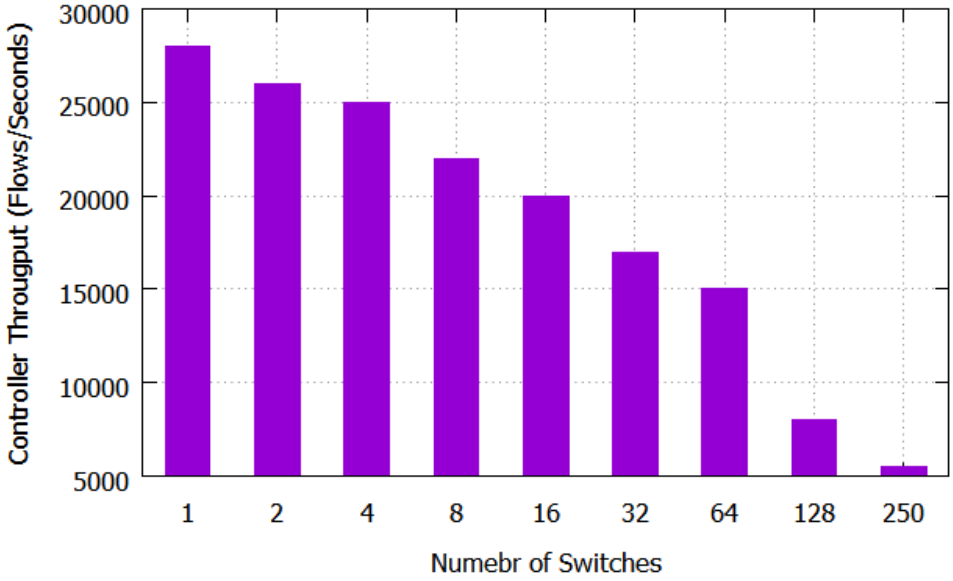

Figure 6.2: Throughput vs number of switches.

number of requests per second. In a typical network it is very unlikely that all the switches are highly active at the same time, so the controller should be able to manage a far larger number of switches.

POX Controller Response Time

The POX controller response time is an important factor because it directly impacts flow completion times. In this section, we use oftrace [104] to perform our evaluations. Oftrace is a suite of tools that anlayze OpenFlow packects. Oftrace takes as an input a libpcap formated file (from tcpdump/wireshark) and outputs useful statistics about the OpenFlow session.

Oftrace includes two tools ofdump and ofstats. Ofdump simply lists OpenFlow message

```
# ofdump OneSwitch.pcap 192.168.1.5 6637

DBG: tracking NEW stream : 192.168.1.5:6637-> 192.168.1.6:47598

DBG: tracking NEW stream : 192.168.1.6:47598-> 192.168.1.5:6637

FROM 192.168.1.5:6637           TO  192.168.1.6:47598   OFP_TYPE 0
LEN 8   TIME 0.000000

FROM 192.168.1.6:47598          TO  192.168.1.5:6637    OFP_TYPE 0
LEN 8   TIME 0.026077

FROM 192.168.1.5:6637           TO  192.168.1.6:47598   OFP_TYPE 5
LEN 8   TIME 0.029839

FROM 192.168.1.6:47598          TO  192.168.1.5:6637    OFP_TYPE 6
LEN 128 TIME 0.1070415

Total OpenFlow Messages: 20010
```

Figure 6.3: ofdump output.

```
# ofstats OneSwitch.pcap 192.168.1.5 6637

Reading from pcap file OneSwitch.pcap for controller 192.168.1.5 on
port 6637

DBG: tracking NEW stream : 192.168.1.5:6637-> 192.168.1.6:47598

DBG: tracking NEW stream : 192.168.1.6:47598-> 192.168.1.5:6637

0.008088        secs_to_resp buf_id=333 in flow 192.168.1.5:6637 ->
192.168.1.6:47598 - packet_out - 0 queued

0.000454        secs_to_resp buf_id=334 in flow 192.168.1.5:6637 ->
192.168.1.6:47598 - packet_out - 2 queued

0.000437        secs_to_resp buf_id=335 in flow 192.168.1.5:6637 ->
192.168.1.6:47598 - packet_out - 1 queued

192.168.1.6:47598 - packet_out - 0 queued
```

Figure 6.4: ofstats output.

types with timestamps by switch/controller pair, while ofstats calculates the controller processing delay, which is the difference in time between a Packet_in message and the corresponding Packet_out or Flow_mod message. A sample output of ofdump and ofstats is shown in Figure 6.3 and 6.4 respectively. We use ofdump and ofstats to evaluate the minimum and

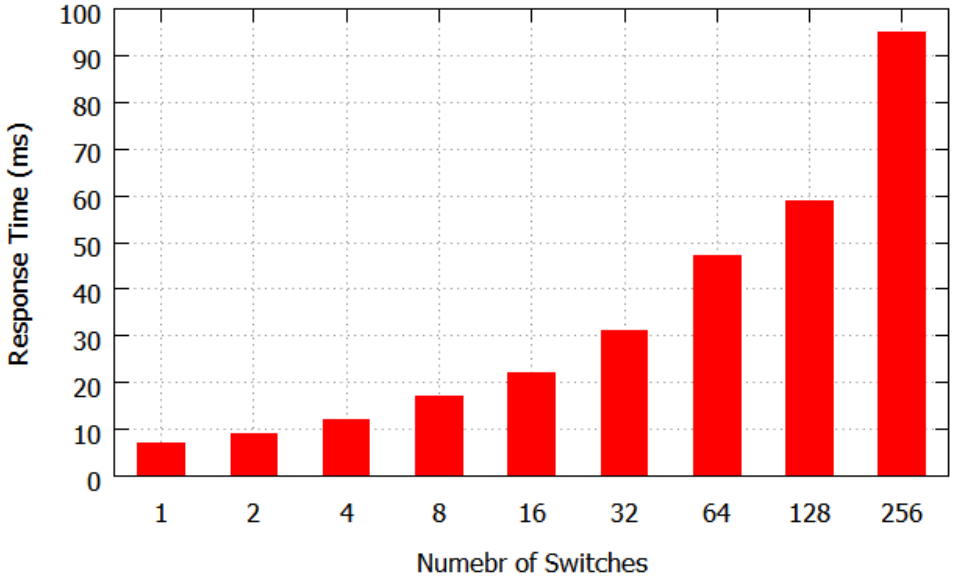

Figure 6.5: Max response time vs number of switches.

maximum controller response time. We also investigate the relation between the controller response time and the number of switches.

For the minimum response time we limit the number of Packet_in messages sent to exactly one. We observe an average response time of 500 microseconds. For maximum response time we simulate a varying number of switches sending as many Packet_in messages as the controller can handle. We observe that the number of Packet_in (load) is inversely proportional to the POX controller response time as well as the number of active switches.

Figure 6.5 shows the maximum POX controller response time versus the number of switches.

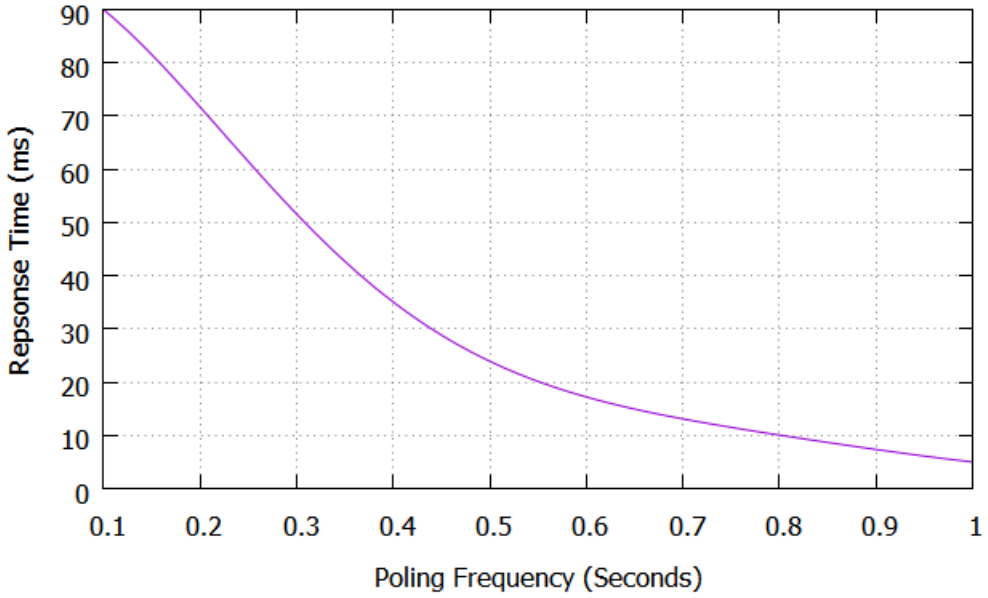

Figure 6.6: Polling frequency vs response time.

Bandwidth consumed by control message sent to the controller

We use ifstat [105] to evaluate the bandwidth consumed by control messages sent to the POX controller. Ifstat is a Linux tool that shows network interface statistics. We stress test the POX controller using cbench and use iftsat to collect bandwidth statistics. We observe a rate of 500 Mbps of Packet_in requests and 150 Mbps of Packet_out messages when the POX controller reaches its full capacity. The rate is asymmetrical because a Packet_in message contains the packet as well as a buffer reference in the switch. The controller only sends back the buffer reference (a small integer) in the Packet_out message.

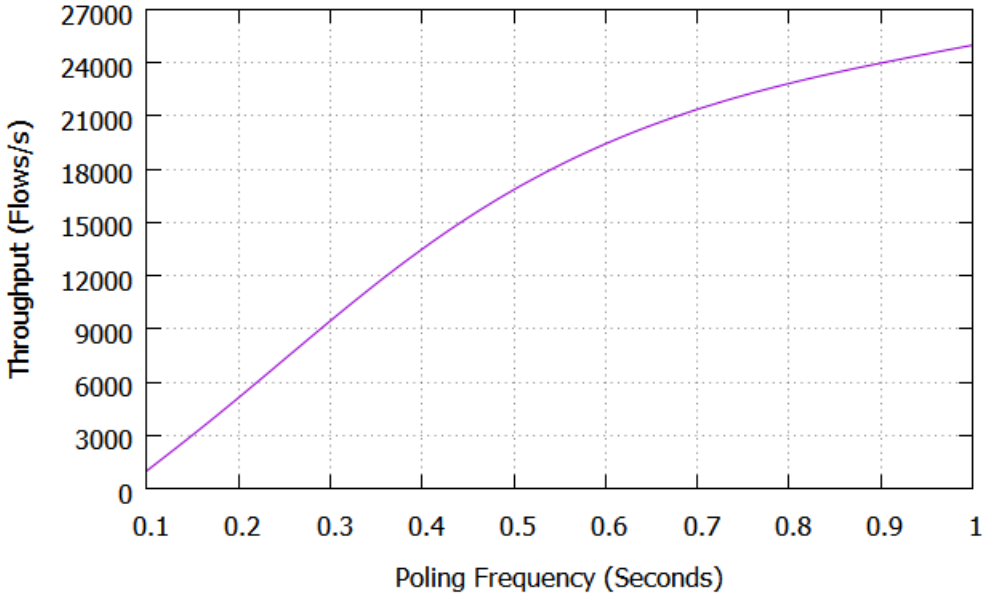

Figure 6.7: Polling frequency vs throughput.

Polling Frequency

The algorithms in OneSwitch use flow statistics gathered from switches to make routing decisions. In order to gather flow statistics the POX controller periodically polls switches. To understand the impact of the polling frequency on the performance of the POX controller. We use cbench to measure the average controller response time and maximum throughput while the POX controller polls switches for flow statistics. We install 100 flow entries on the switch and configure the POX controller to poll statistics from the switches using different polling intervals from 0.1 to 1 seconds as shown in Figures 6.6 and 6.7. We observe that the smaller the polling frequency of collecting statistics, the worse the throughput of the controller and the larger the response time from the controller. Specifically, the throughput

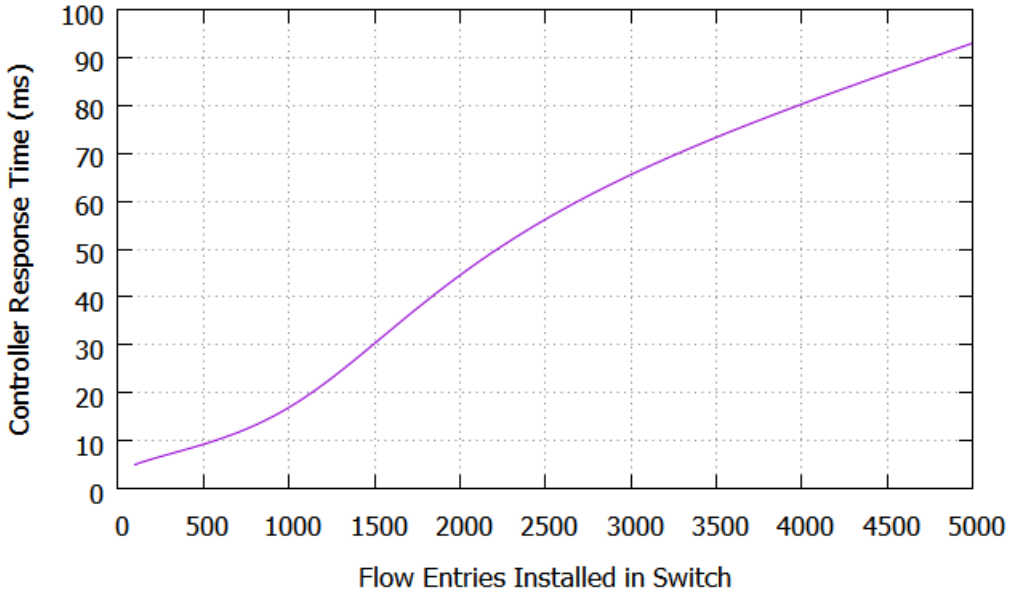

Figure 6.8: Flows installed in switch vs response time.

and response time deteriorate dramatically when the interval decreases below 0.3 seconds. Next we measure the same performance aspects again this time fixing the polling interval to 1 second and varying the number of flows installed on the switch. Figures 6.8 and 6.9 shows the controller response time and maximum throughput versus the number of flow entries configured on the switch, respectively.

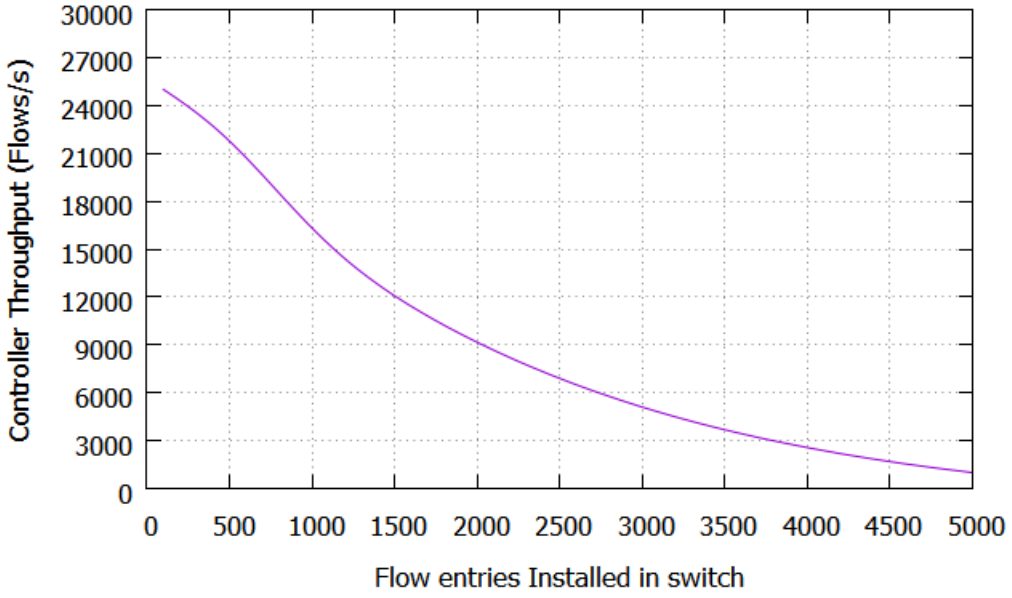

Figure 6.9: Flows installed in switch vs throughput

6.1.2 Location Database

The POX controller in OneSwitch relies on read/write queries to the Location Database to be able to perform ARP resolutions and host-to-switch mappings. The Location Database, once fully converged, will have a number of entries equal to the number of hosts in the network. Each entry consists of 28 bytes, 8 Bytes for DPID address, 12 Bytes for source and destination mac addresses, and 8 Bytes for source and destination IP addresses. The small size of each entry allows the Location Database to scale, in terms of storage capacity, to support large number of hosts. For example a OneSwitch network supporting 10K servers would require less than 1MB of storage.

At any given time there might be a large number of hosts communicating with each other,

requiring the controller to query the Location Database for host-to-switch mappings. The controller has to wait for a response from the Location Database before it can send a response back to the switch.

To obtain a performance baseline we focus on two parameters, throughput and response time. We define throughput, as the number of flows per second the POX controller can support while querying the Location Database for host-to-switch mappings. The response time is defined as the average time it takes for the POX controller to receive a response to host-to-switch mapping queries.

In order to gather performance statistics we use cbench in throughput mode to send flow requests to the POX controller and program the POX controller to query the Location

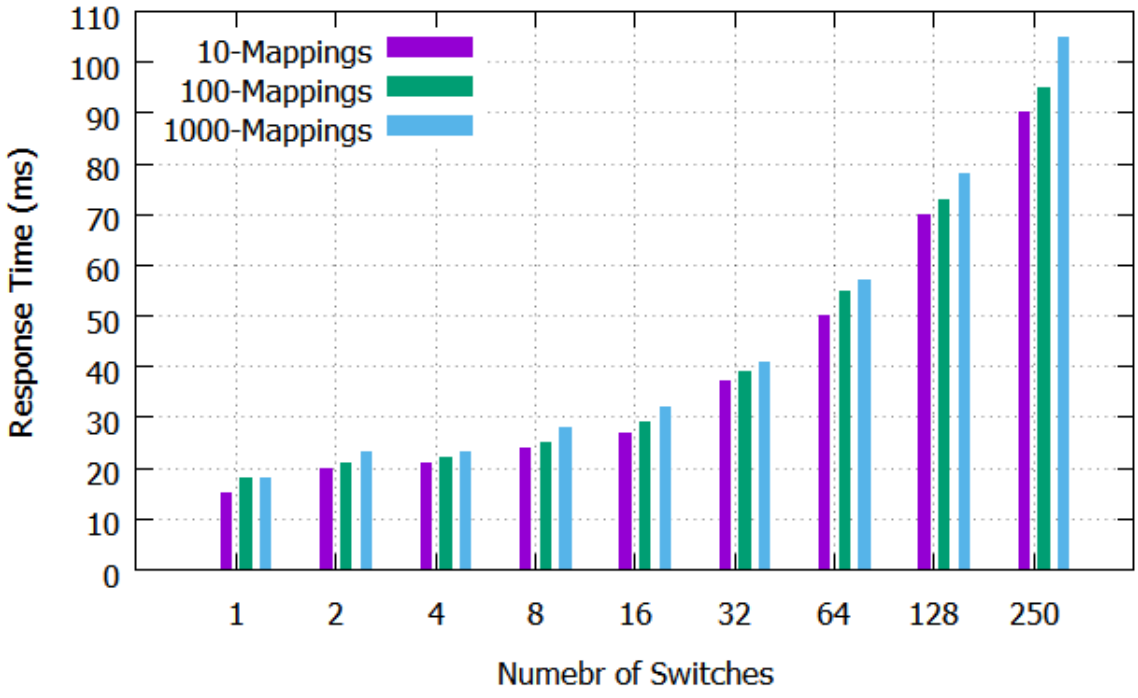

Figure 6.11: Response time vs number of switches.

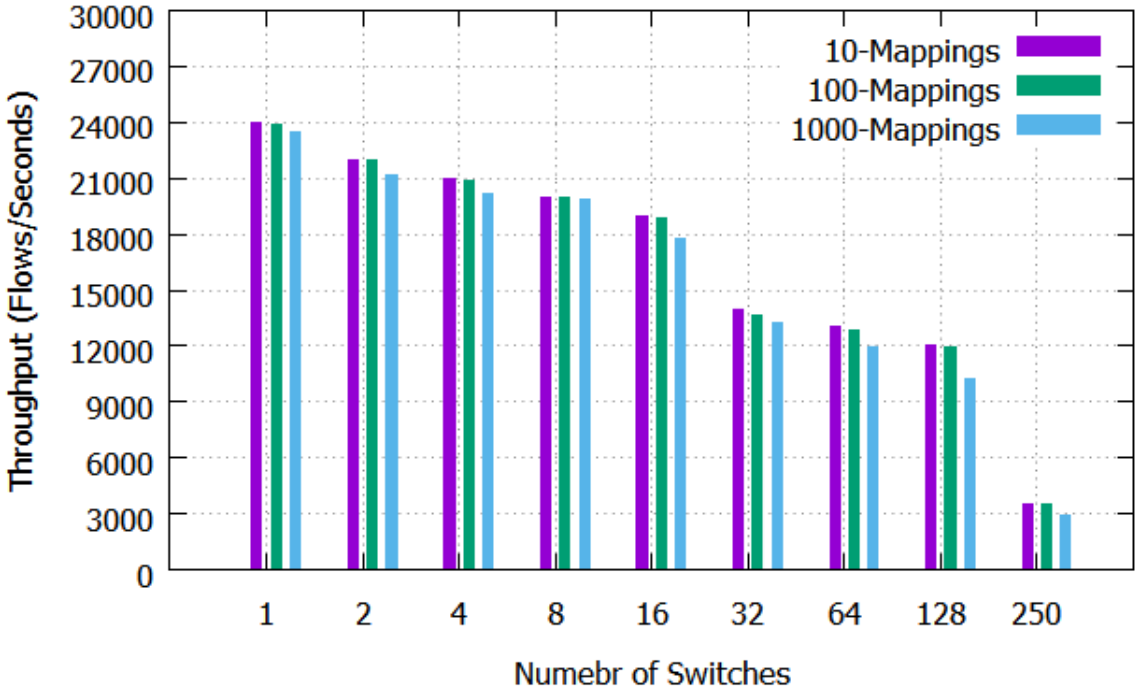

Figure 6.10: Throughput vs number of switches.

Database for host-to-switch mapping before sending a flow_mod response. We pre-populate the Location Database with 10, 100, and 1000 host-to-switch mappings and measure the max queries per second and response time as shown in Figures 6.10 and 6.11, respectively.

6.2 Data Plane Scalability

The data plane in OneSwitch has a number of OpenFlow switches arranged in a folded clos topology. Each OpenFlow switch has a flow table with flow entries shown in Figure 6.12. A flow entry has three parts: Rule field, Stats, and Action. The Rule field is used to define the match condition to an exact flow. Stats are used to count the rule occurrence for

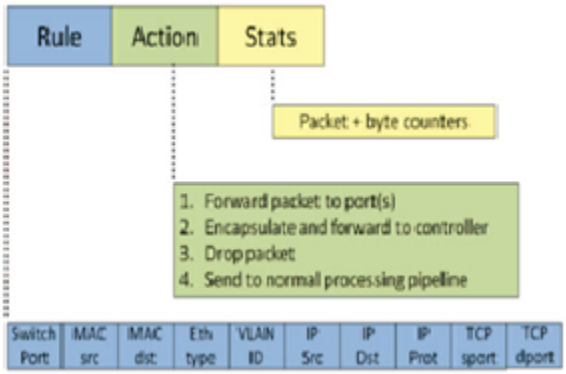

Figure 6.12: Flow Rules

management purposes, and Action defines the action to be applied to a specific flow. When an OpenFlow switch receives a packet to a non existent destination in the flow table, the switch sends the flow to the POX controller to determine how the packets for the flow should be forwarded. The POX controller then configures all the switches along the path from the source to destination. There are 2 main concerns regarding the data plane scalability, the Number of flow rules stored and the time to search rules on a switch.

6.2.1 Number of Flow Rules

The OpenFlow protocol defines two parameters that affect the number of flows stored on a switch. One is Hard Timeout and the other is Idle Timeout. Hard Timeout defines flow entry's maximum lifetime in a switch. The Idle Timeout defines flow entry's maximum idle time in a switch. If the time interval between two packets that belong to a flow entry is more than the Idle Timeout or the Hard Timeout is expired, then the flow entry becomes invalid

and will be removed from the switch.

The flow entry Idle Timeout along with the type of flow pattern directly impact the numbers of flow rules stored in a switch. For example for bursty flows, if the Idle Timeout is set to a large value, then this can lead to many invalid flow entries that can't be removed in time, causing storage capacity to be wasted. On the other hand for non busrty flows, if the Idle Timeout is set to a small value, then this will increase the frequency of invalid flow entries, requiring the involvement of the POX controller in order to program flows.

6.2.2 Time to Search Flow Rules

The Open vSwitches in OneSwitch use Linux based systems to implement OpenFlow switch functions. The Open vSwitches have two types of flow tables. The first one is named linear table and exploits wildcards in the packet header fields to match packets to flows. This table is of small size: it contains only 100 entries. The second table is an exact match table, exploiting a hashing function, and contains up to 131072 entries. The two tables are organized in a chain, and priority is given to the hash table matching.

The forwarding of flows can not happen until the flow table is searched and a match is found (Flow Hit). To analyze the impact of searching the flow table, we use iperf [110] to generate UDP traffic. To asses minimum latency we use a single flow entry that matches the generated traffic. Using a single entry minimizes the table look up cost. For the maximum latency we generate traffic that does not match any of the flow entries and we use a combination of

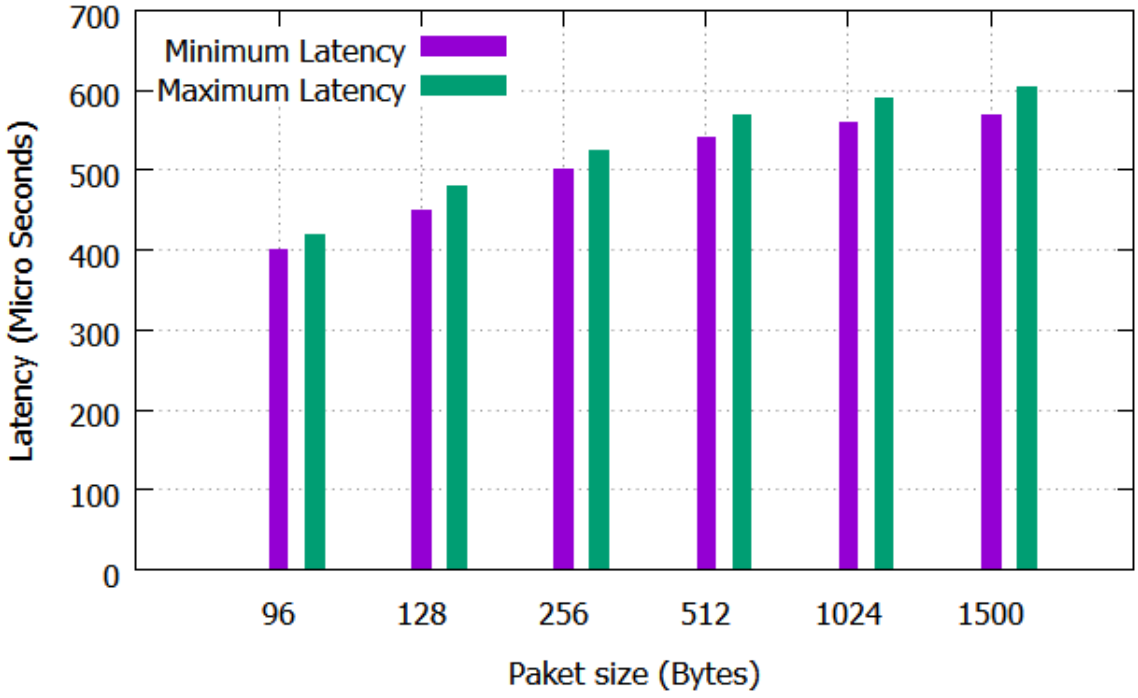

Figure 6.13: Latency due to searching flow rules.

linear table with 100 entries and an Exact Match table with a size of 128K, which is largest size supported by Open vSwitch.

Figure 6.13 shows the minimum and maximum latency for different packet sizes. We observe an average of 5% difference between minimum and maximum latency. Also we observe the latency increases as the packet size increases.

6.3 Routing Service

The Routing Service in OneSwitch uses the BDFS algorithm. BDFS is a graph search algorithm that finds a shortest path from a source node to a destination node in a directed

graph. It searches the shortest path from both the directions: one forward from the starting state, and one backward from the ending state, and stops when the two searches meet in the middle. The Bidirectional search is efficient and fast when used with structured topologies such as Folded Clos topologies.

In the simplified model of BDFS, in which both searches are performed on a tree with a branching factor of b, and the distance from source to destination is d, The complexity of the two searches is represented as $O(b^{d/2})$ and the sum of the time taken to search the two nodes has a complexity of $O(bd)$ less than what would result from a single search from the source to the destination.

In OneSwitch the distance from any source to any destination is always 2 switches away and the the number of search branches is equal to the number of core switches. This implies that BFDS when used with OneSwitch will have a search complexity of $O(c)$ and a search time of $O(2c)$ less than a single search, where c is the number of core switches.

BFDS is used to select the shortest path, but since all destinations, in OneSwitch, have the same cost and are two switches away, BDFS will always return a list of all available paths between source and destination DPID's. A path is then selected from the list based on the load balancing algorithm used.

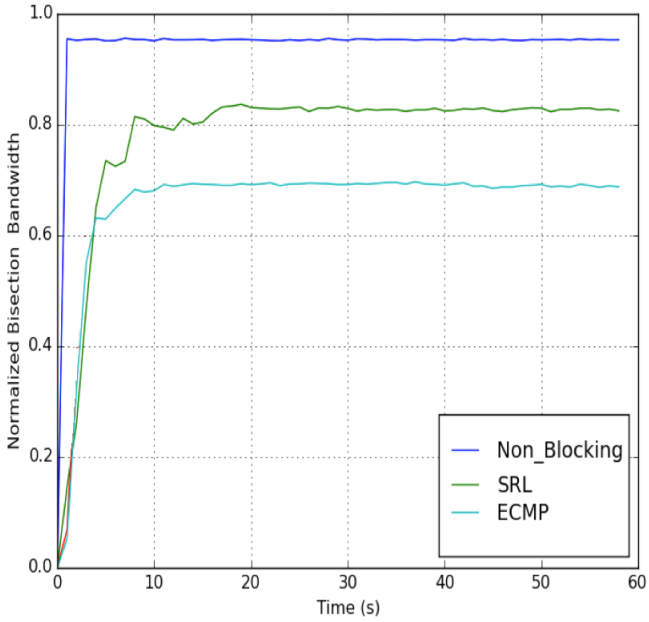

Figure 6.14: K=48, H=512, F=8, and T=Step.

Load balancing and flow optimization algorthims

OneSwitch uses SRL for load balancing, and FlowFit and OFR for flow optimization. In order to compare the performance of these algorithms we implement SRL, OFR and FlowFit as software modules in POX and use OpenFlow to program flow tables in switches.

We utilize Mininet to perform our experiments and testing. The scalability of Mininet depends on the CPU speed and memory size of the underlining machine upon which it is running. We limit the link bandwidth to 10 Mbps, the delay to 1 ms, and the queue depth to 100 packets in order to scale the size of our test bed.

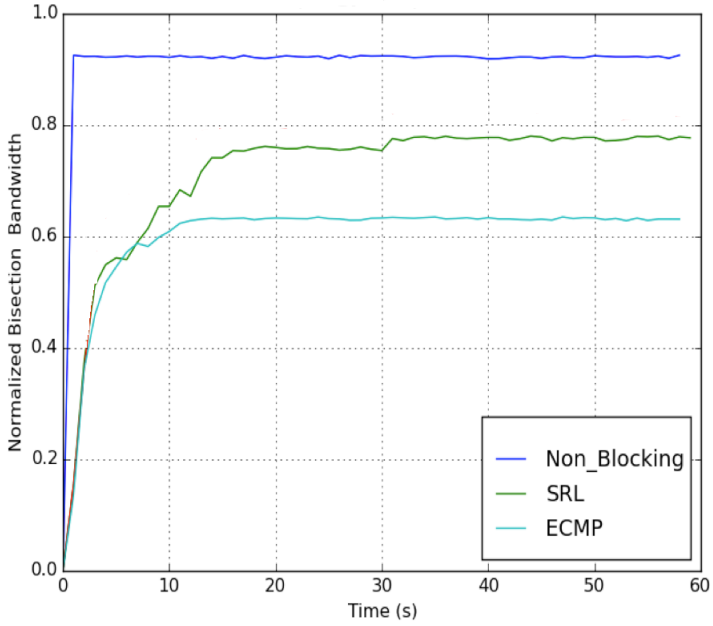

Figure 6.15: K=48, H=512, F=8, and T=Shuffle.

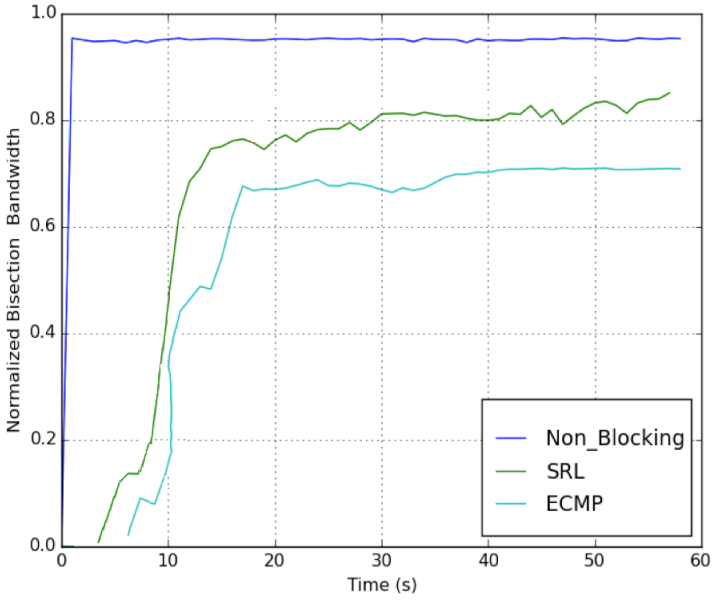

Figure 6.16: K=48, H=512, F=8, and T=Random.

We use Mininet 2.2.1 running on Ubuntu 14.10 instance hosted on Amazon's EC2 cloud platform. The instance type is m4.2xlarge (8 vCPUs, 2.4 GHz, Intel Xeon E5-2676v3, 32 GiB memory). We use Mininet to build a Folded Clos network. The virtual hosts run socket-based sender and receiver programs as applications. We use iperf [110] and sockperf [123] to evaluate performance metrics such as bisection bandwidth, packet loss, and round trip time. We also measure flow completion times for elephant and mice flows with varying number of flows per host.

For elephant flows, hosts send a $1Mbps$ flow that lasts for 10 seconds, every 30 seconds. For mice flows, hosts send a $50Kbps$ flow that lasts for $100ms$, every 1 second. Each virtual host is configured as either a sender or receiver depending on the traffic pattern being emulated. We generate synthetic traffic patterns similar to previous works [38, 58, 93]. We run 5 trials of each experiment. The duration of each trial is 60 seconds. The total experiment run time is 6 hours.

We analyze the normalized bisection bandwidth using a Folded Clos network with $K = 48$ switches, $H = 512$ hosts, $F = 8$ flows/host, and using T =Random, Step, or Shuffle traffic patterns. Figures 6.14, 6.15 and 6.16 compare SRL, the load balancing algorithm used in OneSwitch, to ECMP. Figures 6.17, 6.18, and 6.19 compares flow optimization algorthims, FlowFit and OFR, to ECMP.

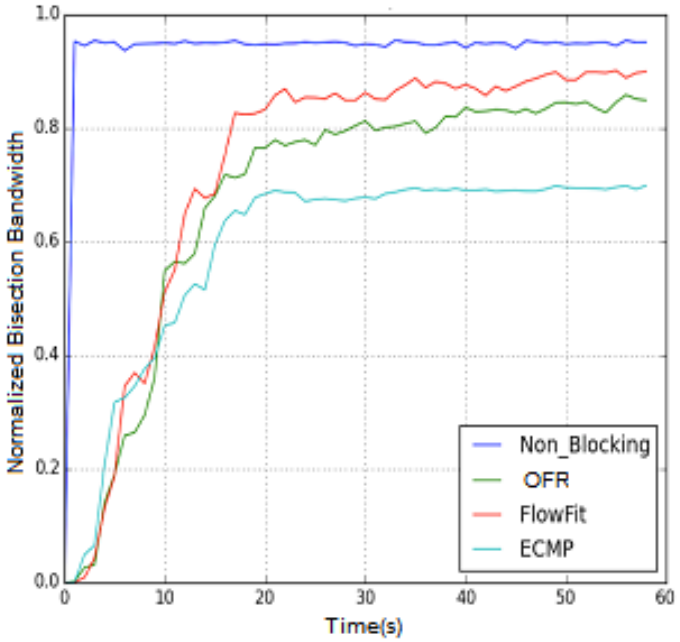

Figure 6.17: K=48, H=512, F=8, and T=Random.

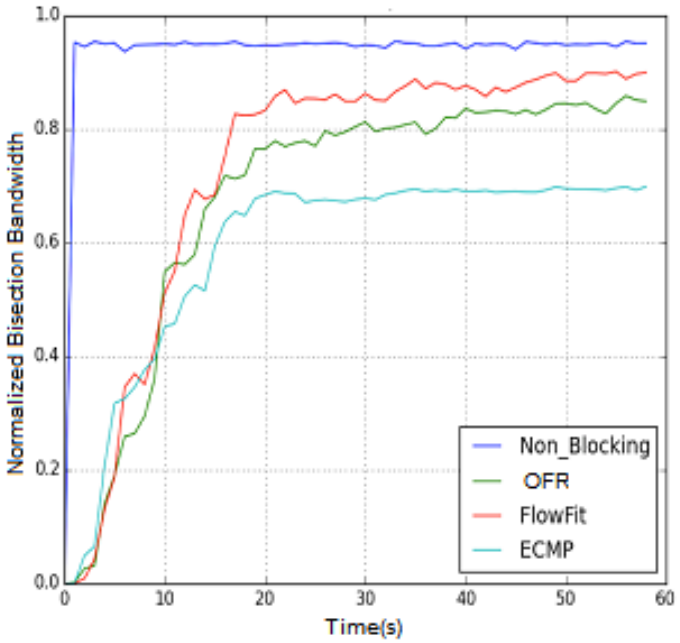

Figure 6.18: K=48, H=512, F=8, and T=Step.

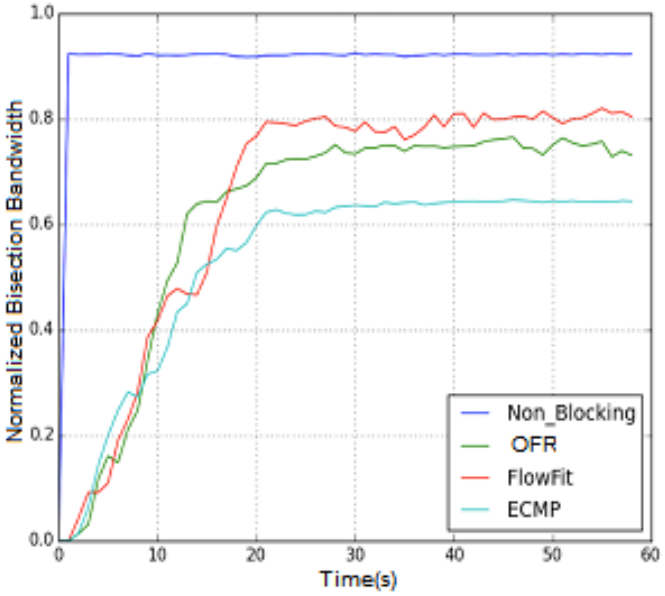

Figure 6.19: K=48, H=512, F=8, and T=Shuffle.

In all scenarios we tested SRL, OFR, and Flowfit outperformed ECMP, with FlowFit achieving the highest normalized bisection, close to 0.90. We also observe an expected increase in normalized bisection bandwidth for all schemes as the number of equal cost paths increases. This is due to path diversity: for the same traffic patterns and flows, there are more paths available to distribute traffic, increasing the overall normalized bisection bandwidth.

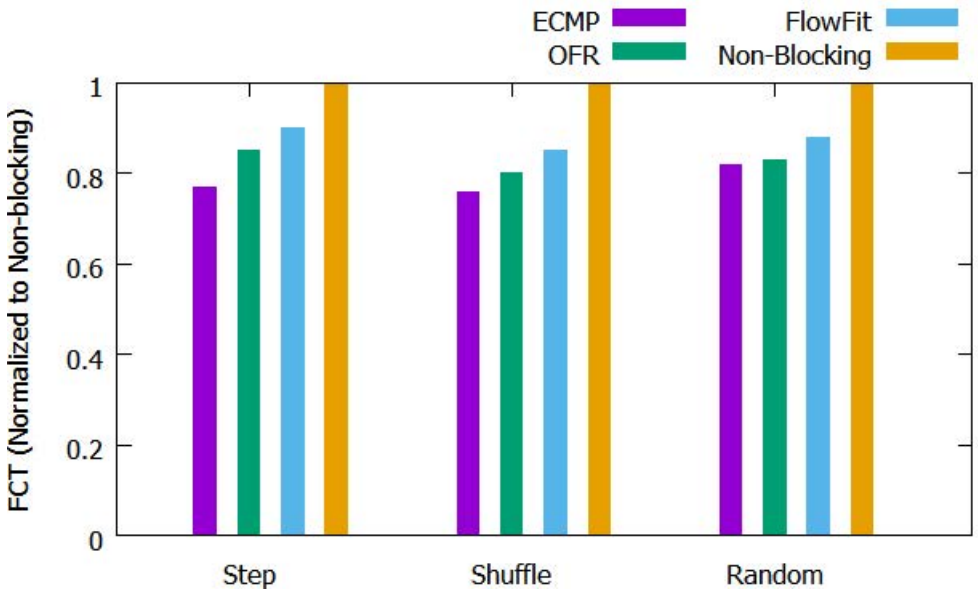

Figure 6.20: Normalized FCT for elephant flows, K=48, H=512, and F=4.

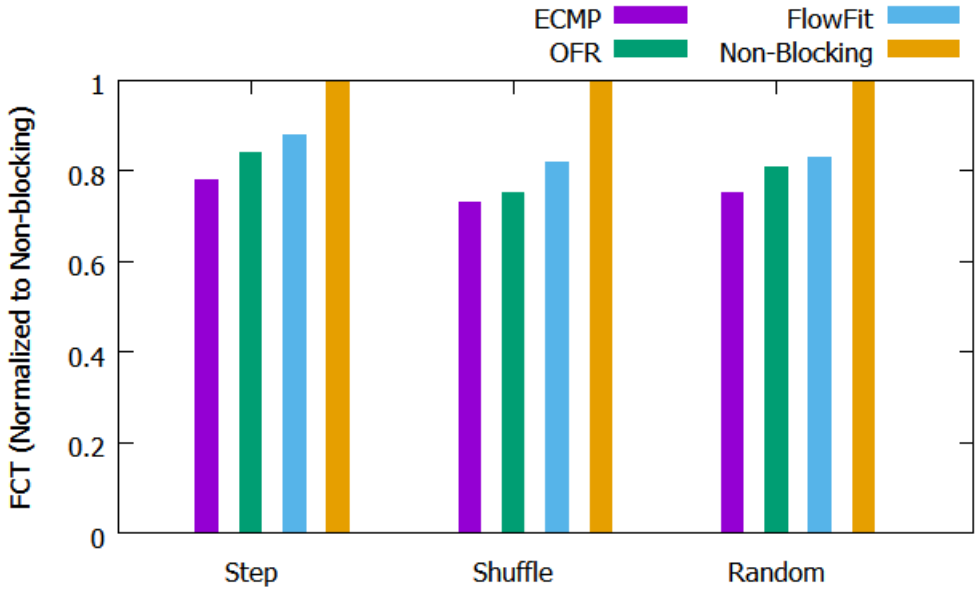

Figure 6.21: Normalized FCT for elephant flows, K=48, H=512, and F=8.

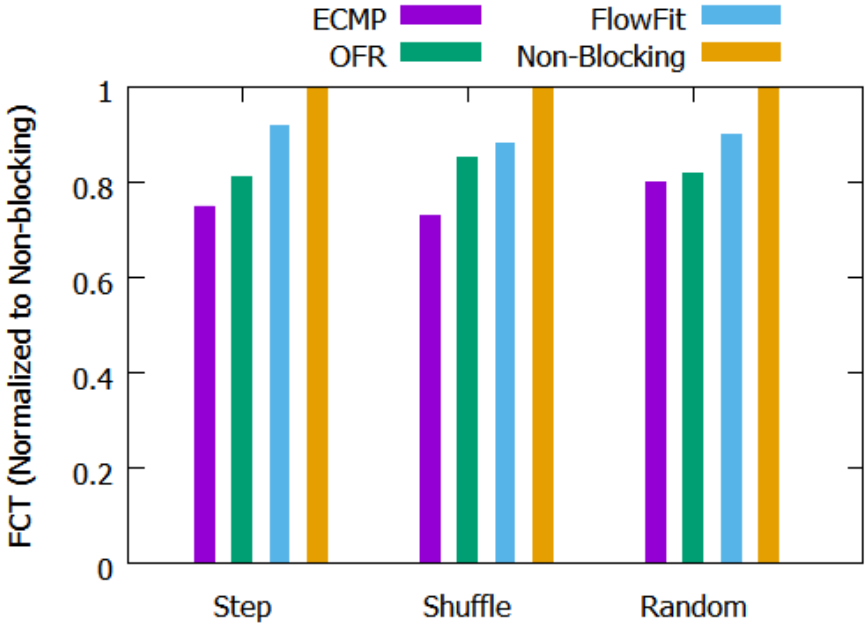

Figure 6.22: Normalized FCT for mice flows, K=48, H=512, and F=4.

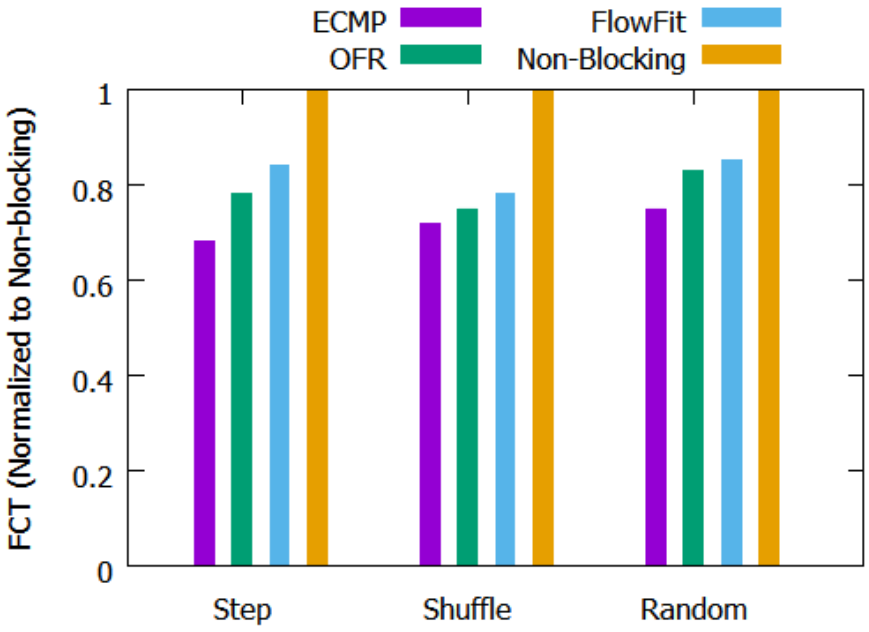

Figure 6.23: Normalized FCT mice flows, K=48, H=512, and F=8.

Figures 6.20-6.23 show the results for flow completion times (FCT) for flow optimization algorthims. The values are normalized to the optimal FCT that is achievable in a non-blocking network. We find that FlowFit and OFR achieved improved FCT when compared to ECMP for both mice and elephant flows. When comparing between mice and elephant flows, we observe that the average improvement for both types of flows is close.

We suspect that the close results are due to the symmetrical properties of the synthetic traffic being simulated. One thing to note here is that although the improvement in FCT for both mice and elephants is close to each other, the performance gains are more significant for mice flow because of their sensitivity to delays. For mice flows, FlowFit has an average of 11% better FCT than ECMP while OFR is around 9%. For elephant flows the average improvement in FCT is 10% and 9% for FlowFit and OFR, respectively.

In our evaluation of load balancing and flow optimization algorthims we used Mininet to emulate scenarios with up to 512 nodes, while maintaining similar results. We were not able to run any tests beyond 512 nodes due to limitations with Mininet. First, Mininet can only run on a single server, which limits the use of multiprocessor capabilities. Second, switches and hosts in Mininet are simulated by a process and a namespace in Linux, which leads to high CPU consumption when emulating a large number of nodes. For emulations beyond 512 nodes, the CPU utilization was consistently above 98%. As this rate, Mininet would run for up to 9 hours and eventually hang. Packet level simulators, such as ns-2, are also not suitable for simulating very large data centers. For example a simulation of a large data center with 2000 nodes, such as in [46] [52], sending at 1Gbps, with an average packet size of

Data Center Role	Data Center Name	Location	Age (Years) (Curr Ver/Total)	SNMP	Packet Traces	Topology	Number Devices	Number Servers	Over Subscription
Universities	EDU1	US-Mid	10	✓	✓	✓	22	500	2:1
	EDU2	US-Mid	(7/20)	✓	✓	✓	36	1093	47:1
	EDU3	US-Mid	N/A	✓	✓	✓	1	147	147:1
Private	PRV1	US-Mid	(5/5)	✓	X	✓	96	1088	8:3
	PRV2	US-West	> 5	✓	✓	✓	100	2000	48:10
Commercial	CLD1	US-West	> 5	✓	X	X	562	10K	20:1
	CLD2	US-West	> 5	✓	X	X	763	15K	20:1
	CLD3	US-East	> 5	✓	X	X	612	12K	20:1
	CLD4	S. America	(3/3)	✓	X	X	427	10K	20:1
	CLD5	S. America	(3/3)	✓	X	X	427	10K	20:1

Figure 6.24: Data Center measurements published by Microsoft Research

1500 would have to process 10 billion packets for a 60 second run, which would potentially require days of run time to generate results. Flow based simulations, as discussed in section 5.4.3, is required in order to simulate large data centers.

6.4 Practicality of OneSwitch

In the previous sections with evaluated OneSwitch by analyzing each of its components. The evaluations were done using benchmarking tools and synthetic traffic. In this section we analyze the practicality of using OneSwitch in a real data center. For a realistic traffic model we use measurements published by Microsoft Research [46]. The measurements were done for three university campus data centers (EDU1-3), two private enterprise data centers (PRV1-2), and five cloud data centers (CLD1-5). Figure 6.24 summarizes the data collected from each data center, as well as some key properties.

Since we are interested in highly active data centers we investigate the measurement for the private enterprise data center. The study of the PRV2 data center, which has 100 switches

supporting 2K servers, found that most flows are small in size ($\leq 10KB$) and a significant fraction of which last under a few hundreds of milliseconds, while the majority of the bytes were in flows that lasted for 10's of seconds. The data center had between 1000 and 5000 active flows 90% of the time. About 20% percent of those flows lasted more than 10 seconds and more than half the bytes were in flows that lasted longer than 25 seconds. In terms of highly utilized links ($\geq$ 70%) they found the exact number varies over time, but never exceeded 25% of the core links.

The above statistics provide us with key insights on how we can expect OneSwitch to perform in such large data centers. First, since more than half the bytes are in flows that last longer than 25 seconds, a monitoring interval for flow optimization in the range of seconds is sufficient for both identifying and re-routing of elephant flows. The monitoring interval in OFR and FlowFit is set to 5 seconds.

Second, if we assume that the average number of active flow is 2,500 and as many as half of the flows that last more than 10 seconds turn out to be elephant flows, then the total number of elephant flows that OFR would re-route is 250. FlowFit would re-route a much smaller number, because we expect the number of elephant flows on congested links to be much smaller than the total number of elephant flows on all links, especially since the number of congested links never exceeds 25% of the total number of core links.

Third, from our simulations we know that the run time for OFR and FlowFit to compute the re-routing of 250 flows is below 100ms. Forth, we can expect a polling throughput of $\approx 7MB/s$ (72B/flow * 5K flow/switch * 100 switches), and a storage size of $\approx 36MB$ (72B/flow

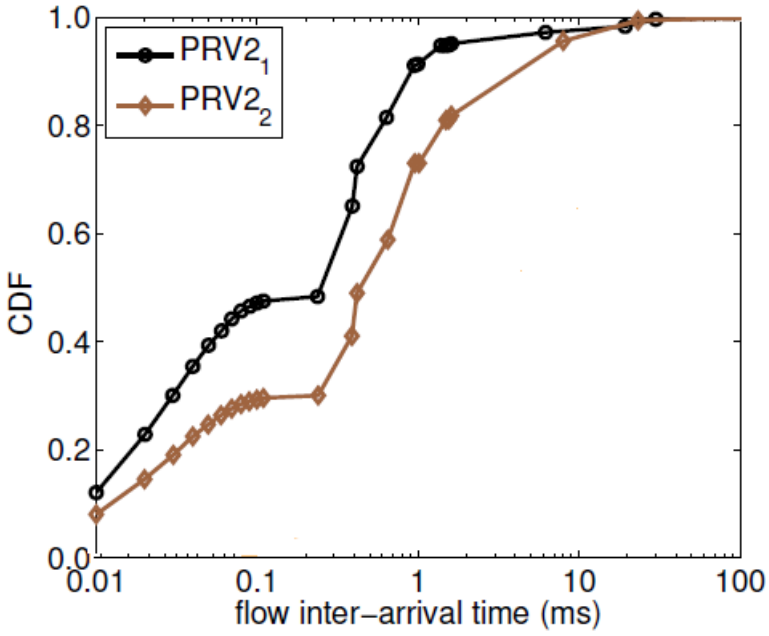

Figure 6.25: Inter-Arrival Time

5K Flows/switch 100 switches) for tracking flows.

The PRV2 data center has small inter arrival rates as shown in Figure 6.25. The flow inter-arrival times have important implications on the scalability of OneSwitch. This is because each edge switch must forward the first packets of each flow to the controller for processing in order to compute a route for each flow. For example, flows with inter-arrival times of 10μ seconds arriving across 100 edge switches would cause an OpenFlow controller to receive 10 million flows per second and edge switches with potentially have a table size of 100K flows. This is far more than the control plane and data plane of OneSwitch can keep up with, as the POX controller can only handle around 30K flows and Open vSwitches support flow table sizes of 128K entries.

The flow duration and size also have implications for the centralized controller. The lengths of flows determine the relative impact of the latency imposed by a controller on a new flow. For example, from the PRV2 data center traffic trace we can see that most flows last less than 100ms. We also know from our benchmarking tests that the average response time is approximately 10ms, this imposes a 10% delay overhead. The additional processing delay may be acceptable for some traffic, but might be unacceptable for other kinds.

In summary, OneSwitch with a single POX controller would only be able to support EDU1-3 data centers but not PRV1-2 nor the five cloud data centers, CLD1-5. To be able to support such large data centers the load could be split across multiple controllers so that the controller does not become the bottleneck of the entire system. Given today's server architecture, we can realistically expect a pool of Openflow controllers to handle large number of connections. Current generation Intel Xeon class servers have the ability to support 2 or more processor sockets with up to 16 cores per socket, providing massive amounts of processor cycles.

6.5 OneSwitch compared to other schemes

In this section we compare OneSwitch to other proposed schemes for the data center. We compare OneSwitch to, MPTCP [75], Hedeara [58], TinyFlow [95], RepFlow [94], CONGA [91], RPS [88], XMP [86], VL2 [48], Portland [53], Diverter [47], NetLord [73], VICTOR [50], Oktopus [65], SecondNet [59], SeaWall [78], GateKeeper [77], NetShare [62], SEC2 [60], and CloudNaaS [66]. These schemes have varying and overlapping features. We select,

main objective, packet forwarding, size of forwarding table, load balancing, packet delivery, reaction to congestion, targeted flows, data plane topology, control plane, fault tolerance, mobility, and implementation as points for comparison. Table 6.1 provides a summary of the comparisons between OneSwitch and the rest of the schemes.

6.5.1 Main Objective

OneSwitch offers a number of important services for the data center, but it's main objective is to provide increased bisection bandwidth for all flows, reduced mean FCT for short flows, easy deployment, and flexible resource allocation. OneSwitch is unique when compared to other schemes, because it combines all these services in one platform. For example schemes such as such MPTCP, Hedeara, and RPS focus only on providing high bisection bandwidth, while schemes such TinyFlow, RepFlow, CONGA, XMP focus on reducing the mean FCT of short flows. On the other hand schemes such as Diverter, Netlord, VICTOR, VL2, Portland, SecondNet Seawall, Gatekeeper, NetShare, CloudNaas address ease of deployment and flexible resource allocation through multi-tenancy solutions and virtualization.

6.5.2 Packet Forwarding

OneSwitch, VL2, Portland, Diverter, NetLord, and SEC2 all use forwarding methods that separate hosts from their location allowing for mobility and easy deployment. For example OneSwitch uses mac address to DPID mappings, VL2 uses IP-in-IP, Portland uses Hierar-

chical Pseudo Mac addresses, Diverter uses Mac re-writing, NetLord uses Mac-in-IP, and SEC2 uses Mac-in-Mac. The rest of the methods use IP prefixes for forwarding, which limits ease of deployment and mobility.

6.5.3 Size of Forwarding Tables

OneSwitch uses host to switch mappings to identity hosts and their location. The mappings are stored in the Location Database. The mappings reduce the size of forwarding tables because switches only have to store addresses of hosts that are directly connected, therefore the size of OneSwitch'es forwarding table is proportional to the number of edge switches. VL2, Netlord and Sec2 are similar in that they use packet encapsulation to maintain forwarding state for only edge switches. These schemes offer the most scalability in terms of size of forwarding tables.

The size of Portland's forwarding tables are proportional to the number of Pod's, Diverter's to the number of physical machines, VICTOR's to the number of VM's, Oktopus to number of neighbors, while Second-Net, Seawall, and Gatekeeper keep states at end-hosts (e.g., hypervisors). These schemes offer good scalability.

The size of forwarding table in MPTCP, Hedeara, TinyFlow, RepFlow, CONGA, RPS, XMP, and CloudNaas is proportional to the number of IP prefixes, which is not scalable and can lead to forwarding table exhaustion in large environments.

6.5.4 Load Balancing

Data centers utilize multiple paths between any pair of end hosts to provide load balancing. OneSwitch uses SRL to load balance traffic across multiple paths. SRL reduces flow collisions by randomly assigning flows to links that are least loaded, as explained in section 4.5.

MPTCP, Hedera, TinyFlow, RepFlow, Portland, XMP, and VL2 all rely on ECMP for load balancing. MPTCP divides a TCP flow into several subflows using multiple address pairs (e.g., IP or port number) for the same physical source or destination servers. TinyFlow segments long flows into short flows and randomly varies the egress port for these flows, RepFlow applies per-flow load balancing by creating another TCP connection for replicated flows (i.e., for the ones smaller than 100 Kbytes) between the sender and receiver. All of the aforementioned methods rely on ECMP which does not take into consideration congestion on links.

CONGA uses Flowlet [36] forwarding for load balancing. RPS randomly selects the egress port from the equal-cost paths for each packet to balance the load on different paths. If RPS paths have a large variance in latency then this can lead to out of order packet, which can degrade performance.

Diverter, NetLord, VICTOR, Oktopus, SecondNet, Seawall Gatekeeper, SEC2, and NetShare do not deploy any type of load balancing and rely on the underlining routing protocols being used to provide load balancing services.

We use Mininet to compare the performance of SRL to MPTCP and Flowlet. We use

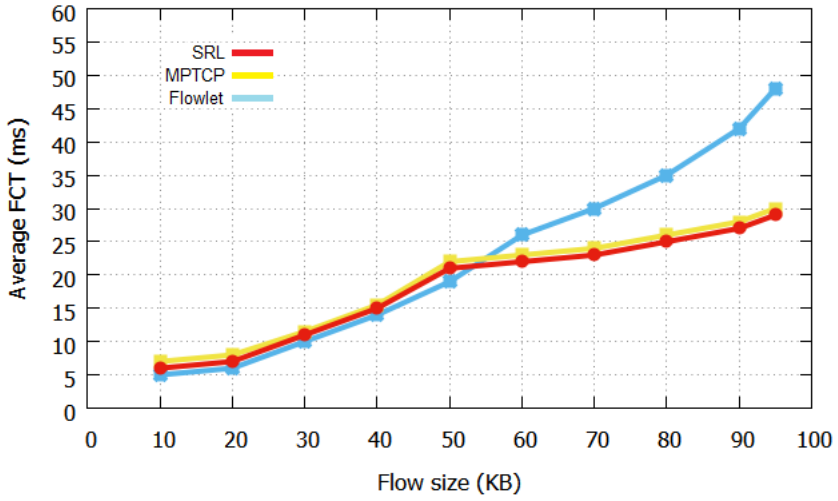

Figure 6.26: Average FCT for SRL, MPTCP, and Flowlet.

MPTCP v0.89, which is implemented in Linux kernel 3.18. We implement Flowlet using hash-based TCP source ports on the outer packet header, except that the hash changes for each new flowlet. This involves taking a hash of the 6-tuple that includes the flow's 5-tuple plus the Flowlet ID (which is incremented every time a new flowlet is detected at a switch).

We simulate a simple client-server communication model where each client chooses a server at random and initiates a number of persistent TCP connections. We vary the flow size for each test and measure the average FCT as shown in Figure 6.26. We observe that for small flows the average FCT for all methods is close. As flows sizes increase the avergae FCT dramatically increases for Flowlet, while continues to increase at a steady and close rate for both SRL and MPTCP, with SRL achieving slightly better average FCT. We also observed that the MPTCP implementation incurred high CPU utilization and unstable results when running more than 4 flows per host between two end-points.

6.5.5 Reaction to Congestion

OneSwitch uses FlowFit to proactively minimize congestion. FlowFit periodically reassigns elephant flows on overloaded links to underloaded links in an effort to control congestion. Hedera is similar to OneSwitch in that it uses a central controller to monitor links and flows, but uses a Simulated Annealing algorithm to assign flows to links.

TinyFlow and RepFlow depend on ECMP, which is ineffective in handling congestion in the presence of large persistent flows.

CONGA reacts to congestion by performing a table look-up on collected congestion measurements at edge switches to select the egress port to forward flowlets.

XMP reacts to congestion by generating ECN messages when the instantaneous queue length of any switch exceeds a threshold to inform the sender to adjust its congestion window. MPTCP reacts to network congestion by linking TCP's congestion-avoidance algorithm on multiple subflows and by explicitly relocating the subflows on more congested paths to less congested ones. Diverter, NetLord, VICTOR, Oktopus, SecondNet, Seawall Gatekeeper, SEC2, and NetShare do not deploy any mechanisms to react to congestion.

We use Mininet to compare the performance of FlowFit to Presto and Hedera. Authors of CONGA present impressive results for bisection bandwidth and FCT, but unfortunately we are unable to compare it to FlowFit because the table look-up on collected congestion measurements in CONGA is implemented using custom ASIC's.

The implementation of Hedera is available on Github [108]. As for Presto we implement

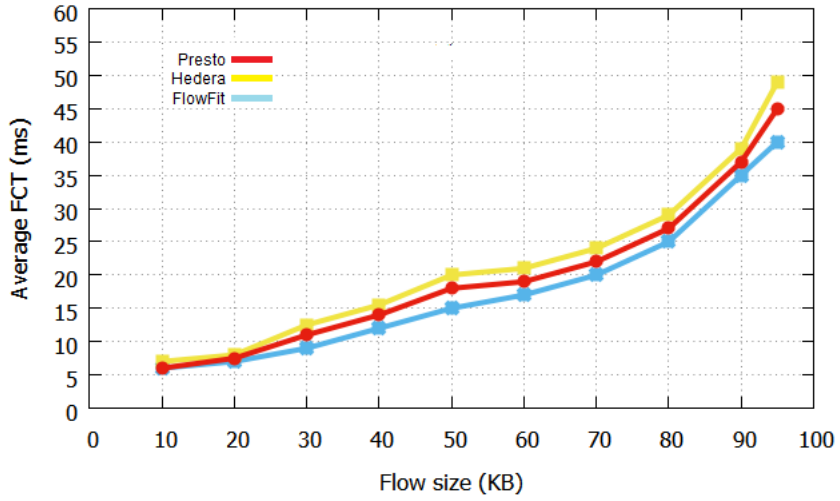

Figure 6.27: Average FCT for FlowFit, Hedera, and Presto.

our own version as we could not find any readily available implementation. We implement Presto in vSwithces. We use the POX controller for configuring multiple spanning trees and shadow-MAC-based forwarding in the dataplane as described in [98]. The encapsulation header contains the flow ID (hash of the 5-tuple) and flowcell ID. The flow and flowcell ID are used to implement out of order resembley at the receiver side. Although our implementation of Presto is not identical to the paper, it faithfully reproduces the core elements.

We run our tests using the same simple client-server communication model used in the previous section with varying flow sizes. Figure 6.27 shows the average FCT for FlowFit, Hedera, and Presto. We observe very close results with FlowFit achieving the better results as flow size increases.

6.5.6 Targeted Flows

OneSwitch, Hedera, TinyFlow, and XMP only targets elephant flows when performing re-routing, which leads to increased bisection bandwidth for elephant flows as well as improving FCT for mice flows.

RepFlow targets only short flows, while MPTCP, CONGA, and RPS target all flows.

Diverter, NetLord, VICTOR, Oktopus, SecondNet, Seawall Gatekeeper, SEC2, and NetShare do not target any flows.

6.5.7 Data Plane Topology

OneSwitch uses a Folded Clos topology for it's data plane. The Folded Clos topology provides equidistant bandwidth and latency between any of its connected hosts. This allows hosts to connect to any edge switch and still have the same bandwidth and latency available to them. Since the topology doesn't provide any preference to one host over another, its up to the routing protocols to ensure hosts receive their equal share of the bisection bandwidth. CONGA and VL2 are similar to OneSwitch in that they both use a Folded Clos topology. On the other hand, MPTCP, Hedeara, TinyFlow, and RepFlow all utilize a Fat Tree topology. SecondNet uses a Bcube topology, while Diverter, NetLord, VICTOR, Oktopus, Seawall Gatekeeper, SEC2, and NetShare do not mandate any specific topology, which can lead to unpredictable performance.

6.5.8 Control Plane

OneSwitch uses a centralized OpenFlow controller for control plane operations. One of the main advantages of having a centralized control plane is that is provides a global view of the network state. This allows policy based controls to be built to achieve specific objectives. For example OneSwitch uses a centralized control plane to achieve high bisection bandwidth, mobility, and simplified management.

Hedera, NetLord, VICTOR, Oktopus, SecondNet, SEC2 and CloudNaaS all use a centralized controller. MPTCP is a TCP-compatible scheme; it is a distributed scheme (i.e.,executed by the end hosts) and inherits the basic operations of TCP. TinyFlow, RepFlow, CONGA, RPS, and XMP all use distributed control mechanisms where forwarding decisions are made locally.

6.5.9 Fault Tolerance

Fault-tolerance covers failure handling of components in the data plane (e.g., switches and links) and control plane (e.g., lookup systems). For data plane fault tolerance OneSwitch uses a Folded Clos topology with multiple links, which makes it tolerant to link and switch failures. This is similar to Hedera, MPTCP, CONGA, RPS, and XMP. on the other hand SecondNet uses a spanning tree signalling channel to detect failures, and its allocation algorithm to handle them. NetLord relies on SPAIN [63] agents for fault-tolerance, and VL2 and NetShare rely on the routing protocols such as OSPF.

Diverter, VICTOR, and SEC2 rely on the underlying forwarding infrastructure for failure recovery. Schemes such as Oktopus and CloudNaaS handle failure by re-computing the bandwidth allocation for the affected network. Schemes including Seawall and Gatekeeper can adapt to failures by re-computing the allocated rates for each flow.

As for control plane fault tolrenece, OneSwitch can be used with two centralized Open flow controller in an active/standby setup to protect against single point of failure.

OneSwitch, NetLord, VICTOR, VL2, Portland, SEC2 all include centralized lookup systems for resolving address queries. These lookup systems can be can be deployed on multiple servers in order to improve resiliency.

6.5.10 Mobility

OneSwitch, VL2, Portland, Diverter, NetLord, and VICTOR were the only schemes that offered mobility services, allowing for hosts to move from one location to another with minimal configuration. Mobility and ease of configuration are almost a necessity in large data centers, as configuration tasks can become very consuming and make the data center less agile.

6.5.11 Implementation

OneSwitch is implemented using a POX controller, Location Database, and a Routing Service. Hedera uses a central controller and OpenFlow switches. MPTCP and XMP uses a

modified TCP/IP stack distributed throughout the data center. TinyFlow and RepFlow use OpenFlow switches and do not require any modification to TCP.

CONGA uses custom switching Application-Specific Integrated Circuits (ASICs) to perform its operations, such as flowlet detection, storing congestion measurement tables, load calculation for the links, generation of congestion feedbacks, and congestion marking. RPS does not require a specific implementation detail except regular commodity switches. VL2 is implemented using a software module insatlled on switches and a Location Directory installed on commodity servers.

The main component of Portland is the Fabric Manager. Diverter, NetLord are implemnted as software modules installed on every physical server. VICTOR uses Virtual routers installed on VM's. Oktopus is implemented using a virtual cluster and virtual oversubscribed cluster. The main compnent for SecondNet is the VDC manager. SeaWall is implemnted as a Shim layer deployed as a Network Driver Interface Specification. GateKeeper uses logical switches that connecet vNIC's together. NetShare uses Fulcrum switches. SEC2 uses Forwarding Elements (FEs) and a Central Controller (CC) and CloudNaaS uses a Cloud OpenFlow Controller.

6.5.12 Packet Delivery

Schemes that exploit the multipath feature of DCNs forward the packets of a flow using multiple alternative paths between source and destination end hosts. However, this technique

may lead to delivering packets out-of-sequence because of the latency differences (stemming from their different queueing delays) of alternative paths.

Out-of-sequence packet delivery occurs when packets with higher sequence numbers arrive before those with lower sequence numbers, which left the sender earlier. In that case, TCP may unnecessarily retransmit delayed out-of-sequence packets

OneSwitch and Hedeara reassign elephant flows, when congestion occurs, to a new path that is selected among the equal-cost paths to the destination. Because the relocation of the flow is carried out by selecting a new path, out-of-sequence packet delivery is unlikely.

Out-of-sequence packets may be experienced in MPTCP as each subflow may take a different path, and the selection of paths in MPTCP is oblivious to the path's latency. MPTCP then apportions a measure against out-of-sequence packets; each segment carries two different sequence numbers in its header. The first one is the connection-level sequence number, and the second one is a subflow-specific sequence number. By mapping these two sequence numbers, the receiver reassembles the original byte stream [75].

TinyFlow employs OpenFlow-based edge switches to randomly vary the egress port of a long flow when 10KB of data have been sent. In this way, TinyFlow segments long flows into 10KB flows. The selection of a new egress port for the 10KB segments is performed among the equal-cost paths to the destination. However, some packets of these segmented flows may be delivered out-of-sequence.

RepFlow is a multipath forwarding scheme, that does not suffer from out-of-sequence packet

delivery because it employs a flow-based forwarding technique; every packet of each replicated flow follows the same path to the destination [50].

CONGA does not suffer from out-of-sequence packet delivery as long as the inter-flow gap is large enough (i.e., if the idle time between two consecutive bursts of a flow is larger than the maximum latency difference among the multiple paths to the same destination). Moreover, a flowlet inactivity timeout parameter introduces a compromise between the number of out-of-sequence packets and the number of flowlets that exploit multiple paths. Therefore, the flowletinactivity timeout must be carefully selected to avoid out of-sequence packet delivery while maintaining a satisfactory level of load balancing [51].

RPS is a packet-based forwarding scheme, which may lead to out-of-sequence packet delivery. The problem is not handled in RPS because it works under the assumption that TCP tolerates some out-of-sequence packets and the paths under RPS are expected to be equally loaded [52].

XMP may also lead to out-of-sequence packet delivery because it forwards subflows using alternative paths, as MPTCP does [53].

Table 6.1: OneSwitch vs other schemes.

Scheme	Main Objective	Packet Forwarding	Load Balancing
OneSwitch	High bisection bw	MAC-to-DPID	SRL
MPTCP	High bisection bw	IP prefixes	ECMP+Virtual addresses
Hedeara	High bisection bw	IP prefixes	ECMP
TinyFlow	Low FCT for mice	IP prefixes	ECMP+Long flow splitting
RepFlow	Low FCT for mice	IP prefixes	ECMP+Replicating short flows
CONGA	Low FCT for mice	IP prefixes	Flowlet forwarding
RPS	High throughput	IP prefixes	Packet Spraying
XMP	Low FCT for mice	IP prefixes	ECMP+Virtual addresses
VL2	Resource allocation	IP-in-IP	ECMP+VLB
Portland	Scalability	Pseudo MAC	ECMP+VLB
Diverter	Layer 3 Virtualization	MAC rewrite	None
NetLord	Layer 2+3 Virtualization	MAC-in-IP	Multiple Spanning Trees
VICTOR	VM migration	IP prefixes	None
Oktopus	Revenue Control	IP prefixes	None
SecondNet	BW guarantee	Source Port	None
Seawall	BW sharing	IP prefixes	None
GateKeeper	Resource utilization	IP prefixes	None
NetShare	BW allocation	IP prefixes	None
SEC2	Secure isolation	MAC-in-MAC	None
CloudNaaS	Manage applications	IP prefixes	None

Scheme	Congestion	Targeted Flows	Data Plane	Control Plane
OneSwitch	FlowFit	Elephant flows	Folded Clos	Centralized
MPTCP	Sub flows	All flows	Fat Tree	Distributed
Hedeara	Simulated Annealing	Elephant flows	Fat Tree	Centralized
TinyFlow	None	Elephant flows	Fat Tree	Distributed
RepFlow	None	Elephant flows	Fat Tree	Distributed
CONGA	Congestion metric	Mice flows	Folded Clos	Distributed
RPS	None	All flows	Fat Tree	Distributed
XMP	subflows + ECN	All flows	Fat Tree	Distributed
VL2	None	Elephant flows	Folded Clos	Distributed
Portland	None	All flows	Fat Tree	Distributed
Diverter	None	All flows	unspecified	Centralized
NetLord	None	None	unspecified	Centralized
VICTOR	None	None	unspecified	Centralized
Oktopus	None	None	Tree	Centralized
SecondNet	None	None	Bcube	Distributed
Seawall	None	None	unspecified	Distributed+Centralized
GateKeeper	None	None	unspecified	Centralized
NetShare	None	None	unspecified	Centralized
SEC2	None	None	unspecified	Centralized
CloudNaaS	None	None	None	Centralized

Scheme	Packet Delivery	Mobility	Implementation
OneSwitch	In-sequence	Yes	Controller+Location Database
MPTCP	In-sequence	No	Modified TCP/IP stack
Hedeara	In-sequence	No	OpenFlow
TinyFlow	Out-of-sequence	No	OpenFlow
RepFlow	In-sequence	No	TCP (multiple sockets)
CONGA	In-sequence	No	Custom switching ASICs
RPS	Out-of-sequence	No	Regular commodity switches
XMP	In-sequence	No	Modified TCP/IP stack
VL2	In-sequence	Yes	Software module+Location Directory
Portland	In-sequence	Yes	Fabric Manager+Multi-Rooted Fat-Tree
Diverter	unspecified	Yes	SW module on every phyiscal server
NetLord	unspecified	Yes	SW on every physical server
VICTOR	unspecified	Yes	Virtual routers on VM's
Oktopus	unspecified	No	Virtual cluster
SecondNet	unspecified	No	VDC manager
Seawall	unspecified	No	Shim layer implemented on NIC
GateKeeper	unspecified	No	Logical switches connecting vNIC's
NetShare	unspecified	No	Fulcrum switches
SEC2	unspecified	No	Forwarding Elements+Central Controller
CloudNaas	unspecified	No	Cloud Controller+OpenFlow

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this work we researched the challenges that data center networks face in supporting virtualization and cloud applications. We listed the main challenges and investigated the suitability of different network architectures in addressing them. We concluded that the folded Clos architecture has nice properties that can be leveraged to address some of the issues, but not all, mainly due to the lack of network state awareness.

We utilized SDN methods to gather and react to network state and proposed the OneSwitch architecture to address the remaining issues, specifically load balancing and flow optimization.

We formulated load balancing in OneSwitch as a Supermarket Problem and presented a prac-

tical version of the problem formulation. We used the practical version of the Supermarket problem to design SRL. SRL uses network state gathered from a OpenFlow controller to perform load balancing decisions. Our Simulation results showed a significant improvement in bisection bandwidth and flow collisions and completion times.

As for flow optimization, we showed through a detailed example how routing in OneSwitch can be formulated as a Binary Multicommodity Flow Problem. We showed through simulation and using a general integer linear programming solver that optimal flow routing without flow collisions, although achievable in polynomial time, is not practical.

We then presented OFR, a modified version of the Binary Multicommodity Flow Problem formulation that optimizes the re-routing of flows by periodically only re-routing elephant flows. The new problem formulation used a heuristics approach to minimize flow collisions.

We also presented FlowFit, a more practical flow optimization algorithm that periodically re-routes only the largest elephant flows on congested links. This allows the optimization to be more efficient because it minimizes the number of elephant flows that need to be re-routed. Our simulations showed that both OFR and FlowFit achieved a significant reduction in flow collisions over ECMP.

We implemented SRL, OFR, and FlowFit in an OpenFlow controller and used Mininet to perform our evaluations and compared the performance of SRL, OFR, and FlowFit to ECMP and a few other proposed algorthims. Our emulation results showed that SRL, OFR and FlowFit significantly increased the bisection bandwidth for elephant flows and reduced FCT

for both elephant and mice flows. The bisection bandwidth and FCT are both important performance parameters for supporting cloud applications that are increasingly becoming more dynamic.

We performed a comprehensive comparison of OneSwitch to a number of other proposed schemes. Our comparison of different schemes revealed several observations. The main observation is that there is no ideal scheme for all the issues that exist in data center networks. This is mainly because data centers have varying requirements that depend on size of the data center and the type of applications being supported. Each data center scheme tries to focus on a few specific requirements. Selecting the best scheme requires a careful understanding of the problem being solved.

The main focus of OneSwitch is achieving high bisection bandwidth and low FCT. Bisection bandwidth FCT are extremely important for big data applications. In that regard OneSwitch outperforms similar schemes such as MPTCP and Hedeara. OneSwitch's superior performance comes at cost to scalabilty. In very large data centers, schemes such as Portland and VL2 are more suitable.

We conclude that in order to support very large data centers we need to increase the throughput of the control plane and the capacity of the data plane. In order to scale OneSwitch's control plane while supporting complex load balancing and flow optimization objectives we must either employ partitioning or parallelism (i.e., use multiple CPUs per controller and multiple controllers), as shown in Figures 7.1 and 7.2.

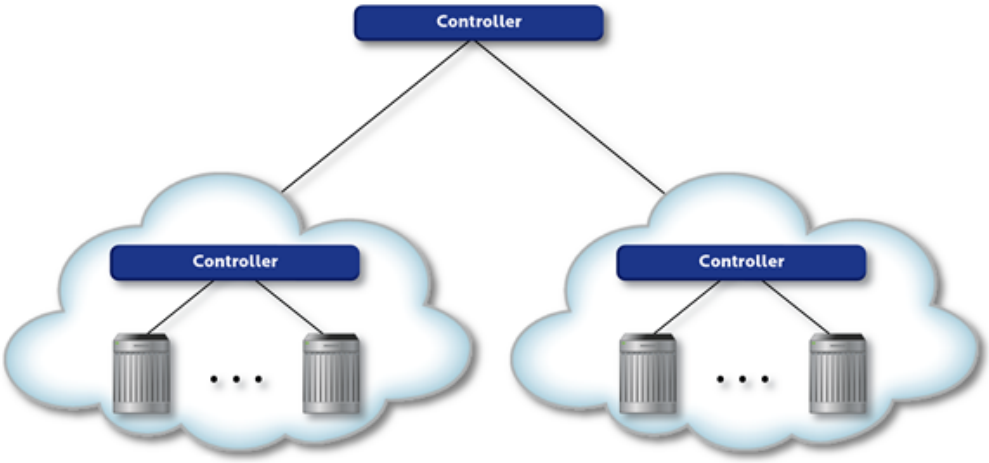

Figure 7.1: Controller partitioning.

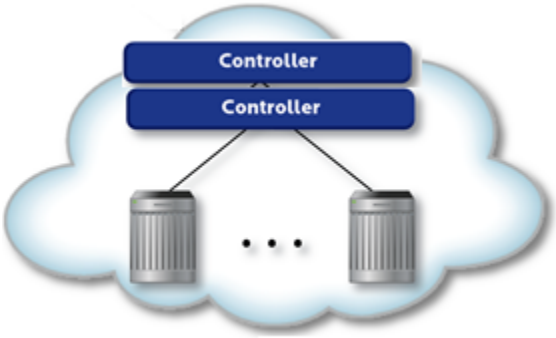

Figure 7.2: Controller parallelism.

In order to increase the data plane capacity, hardware based OpenFlow switches from vendors such as Cisco and HP could could be used [106, 109]. Modern switches support forwarding information bases (FIB) in range of up to 512K entries. With OpenFlow 1.1, a switch can now use these same FIB tables for flows managed by an off-board controller, which alleviates some of the concern about the number of flows a switch can support.

7.2 Future Work

We believe that OneSwitch combines the right balance between computational complexity and performance gains, making it viable option for data centers networks. But as data centers continue to grow in size and as applications vary in performance requirements, further research will be required in order to design control and data plane frameworks that can balance scalabilty versus performance requirements.

Although centralized globally aware data center architectures such as OneSwitch provide excellent performance, they present scalabilty challenges in very large data centers. Further research is needed to design architectures that can scale and keep up with data center demands.

It is very clear that the use of multiple controllers improves the scalability of the control plane, but that opens the question of how to select the mappings between switches and controllers, such that none of these controllers is overloaded. The problem becomes extremely difficult with the dynamic changes of traffic condition.

It is also obvious that the easiest way to scale the data plane is to increase the TCAM capacity. TCAM is an expensive commodity that can be cost prohibitive to implement in very large data centers. Further research is required to design schemes that can effectively improve the utilization efficiency of TCAM capacity. For example future research that comprehensively considers multiple factors such as packet arrival interval, flow rate, or flow type on flow entry Idle Timeout would allow the dataplane to scale while minimizing associated

cost.

Data center traffic is also increasingly volatile and bursty while switch buffers are shallow, thus further research is needed to design centralized control frameworks that can provide more rapid response to congestion (e.g., 10s of microseconds).

Finally, data center architectures must handle asymmetry due to link failures, which have been shown to be frequent and disruptive in data centers, thus future research is needed to design architecture that are robust to asymmetry.

Bibliography

[1] C. Clos. "A Study of Non-Blocking Switching Networks". In: *Bell System Technical Journal* (1953), pp. 406–424.

[2] V. E. Benes. "Mathematical Theory of Connecting Networks and Telephone Traffic". In: *Academic Press* (1965).

[3] V. F. Kolchin, B. A. Sevsat'yanov, and V. P. Chistyakov. "Random Allocations". In: *V. H. Winston I& Sons* (1978).

[4] M. Garey and D. Johnson. "Computers and Intractability, A Guide to the Theory of NP-Completeness". In: *W. H. Freeman and Company* (1979).

[5] Richard Cole and John Hopcroft. "On Edge Coloring Bipartite Graphs". In: *SIAM Journal on Computing 11(3)* (1982).

[6] L. G. Valiant. "A scheme for fast parallel communication". In: *SIAM Journal on Computing* (1982).

[7] L. N. Bhuyan and D. P. Agrawal. "Generalized hypercube and hyperbus structures for a computer network". In: *IEEE Transactions on Computers* (1984).

[8] H. Davis, R. Pollack, and T. Sudkamp. "Towards a better understanding of bidirectional search". In: *In Proc. of AAAI-84* (1984).

[9] R. Melen and J. S. Turner. "Nonblocking Multirate Networks". In: *SIAM J. Comput. Vol.18 No.2* (1989).

[10] S.P. Chung and K. W. Ross. "On nonblocking multirate interconnection networks". In: *SIAM J. Comput. vol. 20 no. 4* (1991).

[11] M Padberg and G Rinaldi. "Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems". In: *Siam Review: 60* (1991).

[12] Tommy Johnson and Bjarne Toft. "Graph Coloring Problems". In: *John Wiley & Sony* (1995).

[13] Cynthia Barnhart et al. "Branch-and-price: column generation for solving huge integer programs". In: *Operations Research Journal* (1998).

[14] S. C. Liew, M. H. Ng, and C. W. Chan. "Blocking and nonblocking multirate Clos switching networks". In: *IEEE/ACM Trans. Netw. vol. 6* (1998).

[15] Curtis Villamizar. "OSPF Optimized Multipath (OSPF-OMP)". In: *IETF Draft* (1998).

[16] K Anstreicher. "Towards a practical volumetric cutting plane method for convex programming". In: *SIAM Journal on Optimization* (1999).

[17] Y. Azar et al. "Balanced Allocations". In: *SIAM Journal of Computing* (1999).

[18] C. Hopps. "Analysis of an Equal-Cost Multi-Path Algorithm". In: *RFC 2992, IETF* (2000).

[19] S. Iyer, A. A. Awadallah, and N. McKeown. "Analysis of a packet switch with memories running slower than the line rate". In: *In Proceedings IEEE INFOCOM* (2000).

[20] Rodeheffer, C. Thekkath, and D. Anderson. "SmartBridge: A scalable bridge architecture". In: *in Proc. ACM SIGCOMM* (2000).

[21] M. Mitzenmacher. "The power of two choices in randomized load balancing". In: *IEEE Transactions on Parallel and Distributed Systems* (2001).

[22] E. Rosen, A. Viswanathan, and R. Callon. "Multi-protocol Label Switching Architecture". In: *RFC 3031* (2001).

[23] C. S. Chang, D.S. Lee, and Y. S. Jou. "Load balanced Birkhoff-von Neumann switches, part I: one-stage buffering". In: *Computer Communications vol. 25 No. 6* (2002).

[24] N. Duffield, C. Lund, and M. Thorup. "Properties and Prediction of Flow Statistics from Sampled Packet Streams". In: *ACM SIGCOMM Internet Measurement Workshop* (2002).

[25] W. Kabacinski and F. Liotopoulos. "Multirate Non-blocking Generalized Three-Stage Clos Switching Networks". In: *IEEE Trans. on Communications, Vol. 50 No.9* (2002).

[26] E. Oki et al. "Concurrent roundrobin-based dispatching schemes for Clos-network switches". In: *IEEE/ACM Trans. Netw., vol. 10* (2002).

[27] A. Smiljani. "Flexible bandwidth allocation in high-capacity packet switches". In: *IEEE/ACM Trans. Netw. vol. 10* (2002).

[28] W. Wang, L. Dong, and W. Wolf. "A distributed switch architecture with dynamic load-balancing and parallel input-queued crossbars for terabit switch fabrics". In: *In Proceedings IEEE INFOCOM* (2002).

[29] N. Duffield, C. Lund, and M. Thorup. "Estimating Flow Distributions from Sampled Flow Statistics". In: *ACM SIGCOMM* (2003).

[30] David Bader, William Hart, and Cynthia Phillips. "Parallel Algorithm Design for Branch and Bound". In: *Tutorials on Emerging Methodologies and Applications in Operations Research. Kluwer Academic Press.* (2004).

[31] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *USENIX OSDI* (2004).

[32] IEEE. "802.1D: Standard for local and metropolitan area networks: Media access control (MAC) bridges". In: (2004).

[33] R. Perlman. "Rbridges: Transparent routing". In: *in Proc. IEEE INFOCOM* (2004).

[34] F. Chang, J.dean, and S. Ghemawat. "Bigtable: A Distributed Storage System for Structured Data". In: *7th USENIX Symposium on Operating System Design and Implementation* (2006).

[35] M. Lasserre and V. Kompella. "Virtual Private LAN Services over MPLS". In: *ietf draft* (2006).

[36] S. Kandula et al. "Dynamic load balancing without packet reordering". In: *In CCR* (2007).

[37] John Kim, William Dally, and Dennis Abts. "Flattened butterfly: a cost-efficient topology for high-radix networks". In: *In ISCA Proceedings of the 34th annual international symposium on Computer architecture* (2007).

[38] M. Al-Fares, A. Loukissas, and A. Vahdat. "A Scalable, Commodity Data Center Network Architecture". In: *In SIGCOMM* (2008).

[39] A. Greenberg et al. "The cost of a cloud: research problems in data center networks". In: *SIGCOMM Computer Communication Review* 39 (2008), pp. 68–73.

[40] C. Guo, G. Lu, and et al. "Dcell: a scalable and fault-tolerant network structure for data centers". In: *In Proceedings of ACM SIGCOMM Computer Communication Review Vol 38.4* (2008).

[41] John Kim, William Dally, and Dennis Abts. "Technology-driven, highly-scalable dragonfly topology". In: *In ISCA Proceedings of the 35th annual international symposium on Computer architecture* (2008).

[42] M. Kim and J. Rexford. "SEATTLE: A Scalable Ethernet Architecture for Large Enterprises". In: *In Proceedings of the ACM SIGCOMM on Data communication* (2008).

[43] J. Naous et al. "Implementing an openflow switch on the netfpga platform". In: *ACM/IEEE ANCS* (2008).

[44] M. Scott and J. Crowcroft. "MOOSE: Addressing the Scalability of Ethernet". In: *In EuroSys Poster session* (2008).

[45] Y. Yu et al. "Dryad: a system for general-purpose distributed data-parallel computing using a high-level language". In: *8th USENIX conference on Operating systems design and implementation* (2008).

[46] T. Benson et al. "Understanding data center traffic characteristics". In: *In Proceedings of the 1st ACM workshop on Research on enterprise networking* (2009).

[47] A. Edwards, F. A, and A. Lain. "Diverter: A New Approach to Networking Within Virtualized Infrastructures". In: *ACM WREN* (2009).

[48] A. Greenberg et al. "VL2: A Scalable and Flexible Data Center Network". In: *In Proceedings of ACM SIGCOMM* (2009).

[49] C. Guo, G. Lu, and et al. "BCube: a high performance, server-centric network architecture for modular data centers". In: *Computer Communication Review Vol 39.4* (2009).

[50] F. Hao et al. "Enhancing Dynamic Cloud-based Services using Network Virtualizations". In: *in Proc. ACM* (2009).

[51] IEEE. "802.1AB: Station and Media Access Control Connectivity Discovery". In: (2009).

[52] S. Kandula et al. "The Nature of Data Center Traffic: Measurements & Analysis". In: *In IMC* (2009).

[53] N. Mysore et al. "PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric". In: *In Proceedings of ACM SIGCOMM* (2009).

[54] M. Alizadeh et al. "Data center TCP (DCTCP)". In: *ACM SIGCOM* (2010), pp. 63–74.

[55] M. Bjorklund. "A Data Modeling Language for the Network Configuration Protocol". In: *RFC 6020, IETF* (2010).

[56] David Breitgand et al. "On Cost-Aware Monitoring for Self-Adaptive Load Sharing". In: *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS* (2010).

[57] P. Costa et al. "CamCube: a key-based data center". In: *Technical Report MSR TR-2010-74, Microsoft Research* (2010).

[58] M. Al-Fares et al. "Hedera: Dynamic flow scheduling for data center networks". In: *In NSDI* (2010).

[59] C. Guo et al. "SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees". In: *in Proc. ACM SOCC* (2010).

[60] F. Hao et al. "Secure Cloud Computing with a Virtualized Network Infrastructure". In: *in Proc. USENIX HotCloud* (2010).

[61] B. Hellerand et al. "Elastictree: saving energy in data center networks". In: *USENIX NSD* (2010).

[62] T. Lam et al. "NetShare: Virtualizing Data Center Networks across Services". In: *Technical Report CS2010-0957* (2010).

[63] J. Mudigonda, M. Al-Fares P. Yalagandula, and J. Mogul. "SPAIN:COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies". In: *in Proc. ACM USENIX NSDI* (2010).

[64] T. White. *Hadoop: The Definitive Guide*. OReilly Media Inc, 2010.

[65] H. Ballani et al. "Towards Predictable Datacenter Networks". In: *in Proc. ACM SIGCOMM* (2011).

[66] T. Benson, A. Shaikh A. Akella, and S. Sahu. "CloudNaaS: A Cloud Networking Platform for Enterprise Applications". In: *in Proc. ACM SOCC* (2011).

[67] T. Benson et al. "MicroTE: Fine grained traffic engineering for data centers". In: *In CoNEXT* (2011).

[68] A. R. Curtis, W. Kim, and P. Yalagandula. "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection". In: *In INFOCOM* (2011).

[69] A. Curtis et al. "DevoFlow: Scaling Flow Management for High-Performance Networks". In: *ACM SIGCOMM,* (2011).

[70] D. Eastlake et al. "TRILL: Transparent Interconnection of Lots of Links". In: *RFC 6326* (2011).

[71] R. Enns. "Network Configuration Protocol". In: *RFC 6241, IETF* (2011).

[72] Greg Goth. "Software-Defined Networking Could Shake Up More than Packets". In: *IEEE Internet Computing* (2011).

[73] J. Mudigonda et al. "NetLord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters". In: *ACM SIGCOMM* (2011).

[74] D. G. Murray et al. "CIEL: a universal execution engine for distributed data-flow computing". In: *8th USENIX conference on Networked systems design and implementation* (2011).

[75] C. Raiciu et al. "Improving datacenter performance and robustness with multipath TCP". In: *ACM SIGCOM* (2011).

[76] A. Rasmussen et al. "Tritonsort: A balanced large-scale sorting system". In: *USENIX NSDI* (2011).

[77] H. Rodrigues et al. "Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks". In: *in Proc. WIOV* (2011).

[78] A. Shieh et al. "Sharing the Data Center Network". In: *in Proc. USENIX NSDI* (2011).

[79] D. Fedyk et al. "IS-IS Extensions Supporting IEEE 802.1aq Shortest Path Bridging". In: *RFC 6329* (2012).

[80] Open Networking Foundation. "OpenFlow Switch Specification Version 1.3.0". In: (2012).

[81] N. Handigol et al. "Reproducible network experiments using container-based emulation". In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012).

[82] Thomas A. Limoncelli. "OpenFlow: A Radical New Idea in Networking". In: *Communications of the ACM* (2012).

[83] D. Lin et al. "Flatnet: Towards a flatter data center network". In: *In IEEE Global Communications Conference (GLOBECOM)* (2012).

[84] A. Tootoonchian et al. "On controller performance in software-defined networks". In: *USENIX Workshop* (2012).

[85] J. Cao et al. "Per-packet load-balanced, low-latency routing for Clos-based data center networks". In: *In CoNEXT. ACM* (2013).

[86] Y. Cao et al. "Explicit multipath congestion control for data center networks". In: *ACM* (2013).

[87] A. Dixit et al. "On the impact of packet spraying in data center networks". In: *In INFOCOM* (2013).

[88] A. Dixit et al. "On the impact of packet spraying in data center networks". In: *INFOCOM* (2013).

[89] D. Erickson. "The Beacon OpenFlow controller". In: *In Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013).

[90] B. Pfaff. "The Open vSwitch Database Management Protocol". In: *RFC 7047, IETF* (2013).

[91] A. Fingerhut et al. "CONGA: Distributed congestion-aware load balancing for datacenters". In: *In SIGCOMM* (2014).

[92] J. Perry et al. "Fastpass: A centralized zero-queue datacenter network". In: *In SIGCOMM* (2014).

[93] J. Rasley et al. "Planck: millisecond-scale monitoring and control for commodity networks". In: *In SIGCOMM* (2014).

[94] H. Xu and B. Li. "Minimizing flow completion times with replicated flows in data centers". In: *INFOCOM* (2014).

[95] H. Xu and B. Li. "TinyFlow: Breaking elephants down into mice in data center networks". In: *IEEE LANMAN* (2014).

[96] M. Armbrust et al. "Spark SQL: relational data processing in Spark". In: *In SIGMOD* (2015).

[97] S. Ghorbani et al. "Micro Load Balancing in Data Centers with DRILL". In: *In HotNets XIV* (2015).

[98] K. He et al. "Edge-based load balancing for fast data center networks". In: *In SIGCOMM* (2015).

[99] URL: `http://mininet.org/`.

[100] URL: `http://openvswitch.org/`.

[101] URL: `https://openflow.stanford.edu/display/ONL/POX+Wiki`.

[102] URL: `http://www.mathworks.com/help/optim/ug/intlinprog.html?requestedDomain=www.mathworks.com`.

[103] *cbench*. URL: `https://github.com/mininet/oflops/tree/master/cbench`.

[104] *cbench*. URL: https://github.com/capveg/oftrace.

[105] *cbench*. URL: https://linux.die.net/man/1/ifstat.

[106] *Cisco OpenFlow Switches*. URL: https://www.sdxcentral.com/cisco/datacenter/definitions/cisco-openflow/.

[107] *Floodlight*. URL: http://www.projectfloodlight.org/floodlight/.

[108] *github*. URL: https://github.com/.

[109] *HP OpenFlow Switches*. URL: https://www.hpe.com/us/en/networking/switches.html.

[110] *iperf: A network performance measurement tool*. URL: https://iperf.fr/.

[111] *Memcached*. URL: https://memcached.org/.

[112] *NEC OpenFlow Switches*. URL: http://www.necam.com/pflow/.

[113] *NOX*. URL: https://github.com/noxrepo/nox.

[114] *OF-Config*. URL: https://github.com/openvswitch/of-config.

[115] *ONOS*. URL: https://onosproject.org/software/.

[116] *OpenDaylight*. URL: https://www.opendaylight.org/.

[117] *OpenStack*. URL: https://www.openstack.org/.

[118] *OSGi*. URL: https://www.osgi.org/business/markets-and-solutions/open-source/.

[119] *proto dhcpd*. URL: https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-proto.dhcpd.

[120] *psutil*. URL: https://code.google.com/p/psutil/.

[121] *pypy*. URL: https://pypy.org/features.html.

[122] *Ryu*. URL: http://osrg.github.io/ryu/.

[123] *Sockperf: A Network Benchmarking Utility over Socket API*. URL: https://code.google.com/p/sockperf/.