Flexplane: A Programmable Data Plane for Resource Management in Datacenters

by

Amy Ousterhout

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

Author ...... Signature redacted ....
Department of Electrical Engineering and Computer Science
August 28, 2015

Certified by... Signature redacted ....
Hari Balakrishnan
Fujitsu Chair Professor
Thesis Supervisor

Accepted by .... Signature redacted ....
Professor Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Flexplane: A Programmable Data Plane for Resource Management in Datacenters

by

Amy Ousterhout

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Network resource management schemes can significantly improve the performance of datacenter applications. However, it is difficult to experiment with and evaluate these schemes today because they require modifications to hardware routers. To address this we introduce **Flexplane**, a programmable network data plane for datacenters. Flexplane enables users to express their schemes in a high-level language (C++) and then run real datacenter applications over them at hardware rates. We demonstrate that Flexplane can accurately reproduce the behavior of schemes already supported in hardware (e.g. RED, DCTCP) and can be used to experiment with new schemes not yet supported in hardware, such as HULL. We also show that Flexplane is scalable and has the potential to support large networks.

Thesis Supervisor: Hari Balakrishnan
Title: Fujitsu Chair Professor

Acknowledgments

Several people have contributed to this thesis over the last two years through their help and support. First I would like to thank my advisor, Hari Balakrishnan. Hari has encouraged me to strive for ambitious goals. He has given me the freedom to work through research problems independently, but has provided hands-on assistance when needed. Jonathan Perry was also very supportive as I was getting started at MIT. I enjoyed working on with him on Fastpass, which became the foundation for this work, and I learned much from him about engineering large systems.

Our collaborators at Facebook were instrumental in both this work and in Fastpass. Petr Lapukhov eagerly devoted several weeks to teaching us how to configure the Facebook switches and helping us run experiments on their clusters. Hans Fugal similarly assisted us with our Fastpass experiments. Omar Baldonado and Sanjeev Kumar provided enthusiastic support of our Facebook collaborations, and Lawrence Brakmo provided insights about experimental design. From Mellanox, Vadim Balakhovski provided generous assistance with Mellanox drivers and equipment.

I would also like to thank Devavrat Shah for helping with the initial design of Flexplane, and Ambika Krishnamachar for prototyping network schemes in Flexplane.

None of this work would have been possible without financial support. The Irwin and Joan Jacobs Presidential Fellowship supported me in my first year and an NSF Graduate Research Fellowship and a Fannie and John Hertz Foundation Fellowship have supported me since then.

I am grateful to all of my officemates and labmates for providing support in the forms of answering questions, reading drafts, or taking breaks from research with me when needed. Thanks Ravi Netravali, Frank Wang, Pratiksha Thaker, Tiffany Chen, Katrina LaCurts, Anirudh Sivaraman, Keith Winstein, Peter Iannucci, and many others.

My friends, both at MIT and elsewhere, have been very supportive throughout my time at graduate school so far. In particular I would like to thank Todd Gingrich for believing in my capabilities, for cheering me up when I needed it, and for giving me an excuse to take occasional much-needed breaks from work.

Finally, my family has always provided support and guidance as I pursued the "family business". Thank you for enabling me to pursue my passions, and for tolerating my decision to attend graduate school on the opposite side of the country.

Contents

1 Introduction **13**

2 Design **17**

2.1 Abstract Packets 18

2.2 Emulation 19

2.3 Accuracy 20

2.4 Flexplane APIs 22

2.5 Scaling the Emulator with Multicore 24

2.6 Fault Tolerance 25

2.7 Deployment 26

3 Implementation **27**

4 Evaluation **31**

4.1 Accuracy 32

4.2 Experimenting with Flexplane 34

4.3 Emulator Scalability 36

5 Related Work **41**

5.1 Software routers 41

5.2 Programmable hardware 43

6 Conclusion and Future Work **45**

List of Figures

2-1 System overview 18

2-2 Emulator pattern API 23

3-1 Source code for a DCTCP queue manager in Flexplane 28

3-2 Source code for a priority scheduler in Flexplane 29

4-1 Latency versus load with one sender 32

4-2 Tradeoffs between resource management schemes 35

4-3 Datacenter network topology and core assignment 36

4-4 Emulator throughput as a function of cores 38

List of Tables

1.1 Comparison with other programmable data planes 14

2.1 API specification 22

3.1 Lines of code for each resource management scheme 28

4.1 Jain's fairness index 33
4.2 Queue occupancies 34

Chapter 1

Introduction

Network resource management schemes promise benefits for datacenter applications. By controlling which packets are transmitted at any given time, they can improve throughput, fairness, burst tolerance, tail latency, flow completion time, and more. To realize these benefits, researchers have developed dozens of different schemes for:

- *Queue management*, i.e., what packet drop or ECN-marking [16] policy should be used? Examples include RED [17], DCTCP [6], HULL [5], and D^2TCP [48].
- *Scheduling*, i.e., which queue's packet should be sent next on a link? Examples include weighted fair queueing (WFQ) [12], stochastic FQ [33], priority queueing, deficit round-robin (DRR) [43], and pFabric [7].
- *Explicit control*, i.e., how should explicit control information in packets be used and modified? Examples include XCP [25], RCP [15, 46], PDQ [21], and D^3 [51].

However, theses schemes can be difficult to deploy on real networks because they require modifications to routers, and building custom ASICs for each new scheme is infeasible for most. This limits the development and adoption of new resource management schemes. Researchers cannot experiment with their nascent schemes in realistic settings. Instead they often rely on simulations (e.g. [20, 32, 49]), which do not capture the impact of other system components, such as CPU utilization or disk I/O, on applications. After schemes have been developed, network operators cannot

Approach	Flexible	Router capacity	Large networks
Software routers	✓	✗	✓
Programmable hardware	✗	✓	✓
Flexplane	✓	✓	?

Table 1.1: Comparison with other programmable data planes.

evaluate them to determine if they provide benefits for their specific applications. In addition, students cannot easily experience implementing protocols and evaluating them in real networks.

A programmable datacenter network could solve these issues. To be effective, it must be *flexible*, support high *router capacity*, and scale to *large networks*. Previous proposals have made progress towards this goal, but ultimately fall short, as shown in Table 1.1. *Software routers* [11, 14, 19, 26, 45] provide flexibility and scale to large networks, but to date have demonstrated significantly reduced router capacity, with the best results providing an aggregate throughput of only 30-40 Gbits/s. *Hardware routers with some programmable elements* [3, 8, 22, 27, 29, 44, 47] run at hardware rates in large networks, but compromise on flexibility.

This paper introduces **Flexplane**, a programmable data plane for datacenters that specifically targets resource management. Flexplane enables users to express their resource management schemes in software using a high-level language (C++), and then deploy them in datacenters with real applications at line rates. It is designed to be flexible while also supporting high router capacity and large networks. We demonstrate that Flexplane already provides the first two properties and argue that, with more work, its design can scale to support large networks as well.

The key idea in Flexplane is to implement the user-specified scheme in a *centralized emulator*, which performs *whole-network emulation* in real time. The emulator, a single designated server, maintains a model of the network topology in software, and users implement their schemes in the emulated routers. Immediately before each packet is sent on the real network, a corresponding emulated packet will traverse the emulated topology and experience the behavior of the resource management scheme in software. In this way, applications observe a network that supports the programmed scheme, though the scheme is implemented in software rather than hardware.

In this paper we present the design of Flexplane and demonstrate that for small topologies (1) it can accurately reproduce the properties of schemes already supported in hardware (e.g. RED, DCTCP) and (2) it can easily be used to experiment with unsupported schemes, such as HULL. We also present preliminary results that suggest that the Flexplane emulator can scale to support large networks; each additional core enables the network to sustain about 20-25 Gbits/s more throughput, up to 10 cores total, which provides 350 Gbits/s.

Chapter 2

Design

Flexplane's goal is to enable a network to support a given resource management scheme at hardware rates, given only a software implementation of the scheme. More specifically, packets should arrive at their destinations with the timings and header modifications (e.g. ECN marks) that they would have in a hardware network that supported the desired scheme.

To achieve this goal, Flexplane implements the scheme in a single dedicated multi-core server called the emulator, which endpoints consult before sending each packet.[1] The transmission of every packet in Flexplane involves three steps (Figure 2-1):

1. *Convey:* At the sending endpoint, Flexplane intercepts the packet before it is sent on the network. It constructs an *abstract packet* summarizing the key properties of the packet and sends it to the emulator (§2.1).
2. *Emulate:* The emulator models the entire network and emulates its behavior in real time. It delays and modifies abstract packets as they would be if run on a hardware network implementing the scheme (§2.2).
3. *Reflect:* As soon as an abstract packet exits the emulation, the emulator sends a response to the source endpoint. The endpoint immediately modifies the corresponding real packet (if necessary) and sends it, reflecting its fate onto the real network (§2.3).

[1]Fastpass [37] previously introduced the idea of scheduling packets in a centralized entity, but Fastpass schedules packets to achieve an explicit objective whereas Flexplane schedules packets to achieve the behavior of a distributed resource management scheme.

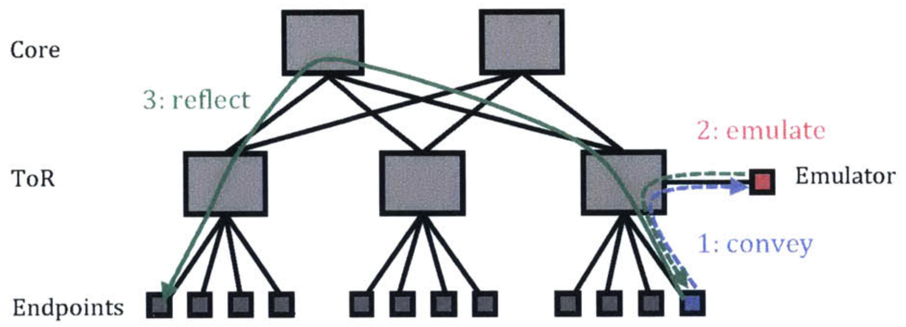

Figure 2-1: In Flexplane, endpoints *convey* their demands to the emulator in abstract packets, the emulator *emulates* the network, and the results are *reflected* back onto the physical network. Dashed lines represent abstract packets; solid line is a real packet.

The result of these steps is that packets experience the network-level behavior in the emulator rather than in the hardware network. While a real packet waits at an endpoint, its abstract packet traverses the emulated topology and encounters queueing and modifications there. When emulation is complete and the real packet traverses the real network, it will encounter (almost) no queueing and no modifications. The effect on higher-layer protocols is that the packet is queued and modified by the emulated resource management scheme.

These steps run in real time, with thousands of packets concurrently traversing the network. Because all packets individually experience the resource management scheme, the result is that datacenter applications experience a network that supports the chosen resource management scheme.

2.1 Abstract Packets

An abstract packet is a concise description of a chunk of data in the same flow to be sent onto the network; it includes all the metadata that the current resource management scheme might need to access at the emulator. By default an abstract packet contains the source and destination addresses of the chunk of data, unique identifier, and flow identifier. In addition, a scheme can include custom information that routers running this scheme would access, such as the type-of-service (differentiated services code point) or whether the sender is ECN-capable. Flexplane provides a general framework that allows schemes to convey arbitrary packet information in abstract packets so that it can support non-standard packet formats (§2.4). For example, in XCP, packets are initialized with a congestion header that includes the sender's congestion window, RTT estimate, and demands. To emulate XCP in Flexplane, the abstract packets include these fields as well.

For simplicity, all abstract packets represent the same amount of data; this can consist of one large packet or multiple smaller packets in the same flow. To reduce CPU and network bandwidth consumed by abstract packets, endpoints and the emulator send abstract packets in batches.

The amount of data represented by each abstract packet is adjustable depending on the user's needs. For precise emulation, an abstract packet should represent exactly one packet in the real network. For better scalability, but at reduced fidelity, an abstract packet could represent multiple packets that belong to the same flow. In this paper, an abstract packet represents one maximum transmission unit (MTU) worth of packets (1500 bytes in our network).

2.2 Emulation

The purpose of emulation is to delay and modify abstract packets just as they would be in a hardware network running the same resource management scheme. To achieve this, the emulator maintains an emulated network of routers and endpoints in software, configured in the same topology as the real network. When abstract packets arrive at the emulator, they are enqueued at their emulated sources. They then flow through the emulated topology, ending at their emulated destinations. As they flow through the emulated network, they may encounter queueing delays, acquire modifications from the routers they traverse (e.g. setting ECN bits), or be dropped, just as they would in a real network. Once an abstract packet arrives at its emulated destination or is dropped, the abstract packet is sent back to the real source endpoint, which modifies and sends the packet, or drops it. The emulator is, effectively, a real-time simulator.

To simplify emulation, the emulator divides time into short timeslots and emulates each separately. The duration of a timeslot is chosen to be the amount of time it takes a NIC in the hardware network to transmit the number of bytes represented by an abstract packet (one 1500-byte MTU by default), so that sending an abstract packet in the emulation takes exactly one timeslot. In each timeslot, the emulator allows each port on each emulated router and endpoint to send one abstract packet into the emulated network and to receive one abstract packet from the emulated network.[2] Timeslots simplify the task of ensuring that the emulation runs in real-time; instead of requiring that each event (e.g. packet transmission) completes at precisely the right time, the emulation only needs to ensure that each timeslot (which includes dozens of events) completes at the right time. This contrasts with ns [20, 32] and other event-based simulators.

2.3 Accuracy

To be a viable platform for experimentation, Flexplane must produce results that match those obtained by a hardware implementation of the same scheme. Thus Flexplane strives for accuracy, meaning that packets experience the same end-to-end latencies in Flexplane as in a hardware network running the same scheme. However, there are a few factors that cause the end-to-end latencies to deviate slightly from this goal.

Consider a network that implements a given resource management scheme in hardware. For simplicity, assume this network consists of a single rack of servers. In this network, the latency l experienced by a packet will be the sum of the unloaded delay u (the time to traverse the network when empty), and the queueing delay q: $l = u + q$. Note that the unloaded delay is the sum of the speed-of-light propagation delay (negligible in datacenter networks), processing delay at each switch, and the

[2]We assume, for simplicity, that all links in the network run at the same rate. If this is not the case, multiple smaller links can be used to represent a single larger link.

transmission delay (the ratio of the packet size to the link rate).

Now consider an identical network that implements the same resource management scheme using Flexplane. The latency l' of a packet in this network consists of the round-trip time r the abstract packet takes to the emulator and back, the emulated transmission delay t_e (the emulator does not model switching or propagation delays), the emulated queueing delay q_e, the time the real packet takes to traverse an unloaded network, and any queueing delay it encounters in the real network, q':

$$l' = r + t_e + q_e + u + q' < 4u + q' + q_e$$

For an emulator in the same rack, $r \leq 2u$ and the emulation ensures that $t_e < u$. Flexplane does not guarantee zero queueing within the hardware network, but in practice we find that q' is very small, approximately 8-12 µs on average (§4.1).

Thus for an unloaded network, the latency in Flexplane is about four times the latency experienced in an equivalent hardware network, or less. Our experiments indicate that in practice this latency overhead is much smaller; in a 10 Gbits/s network, l' is about 33.8 µs compared to 14.9 µs in hardware (§4.1). Furthermore, as queueing within the network increases, the latency overhead of communicating with the emulator becomes less significant, and Flexplane's latencies better match those of hardware.

For topologies with multiple racks of servers, the delay imposed by Flexplane will increase from that described above, but is expected to remain a constant additive factor. With larger topologies, the round-trip time to the emulator and back varies across endpoints. To ensure that all packets experience the same r, the emulator can delay abstract packets from nearby endpoints by a constant amount. It is also possible that q' will increase as the size of the network grows, decreasing the accuracy of the emulation; this must be investigated in future work.

Emulator API - pattern	
int route(AbstractPkt *pkt)	return a port to enqueue this packet to
int classify(AbstractPkt *pkt, int port)	return a queue at this port to enqueue this packet to
enqueue(AbstractPkt *pkt, int port, int queue)	enqueue this packet to the given port and queue
AbstractPkt *schedule(int output_port)	return a packet to transmit from this port (or null)
Emulator API - generic	
input(AbstractPkt **pkts, int n)	receive a batch of n packets from the network
output(AbstractPkt **pkts, int n)	output up to n packets into the network
new(AbstractPkt **pkts, int n)	receive a batch of n packets from the network stack
Endpoint API	
prepare_request(sk_buff *skb, char *req_data)	copy abstract packet data into req_data
prepare_to_send(sk_buff *skb, char *alloc_data)	modify packet headers before sending

Table 2.1: Flexplane API specification. Flexplane exposes C++ APIs at the emulator for implementing schemes and C APIs at the physical endpoints, for reading custom information from packets and making custom modifications to packet headers. An sk_buff is a C struct used in the Linux kernel to hold a packet and its metadata.

2.4 Flexplane APIs

Flexplane exposes simple APIs to users so that they can write their own resource management schemes. It decouples the implemented network schemes from the emulation framework, so that users only need to worry about the specific behavior of their scheme. The framework handles passing abstract packets between different emulated components and communicating with the physical endpoints.

To add a scheme to Flexplane, users can choose between two C++ APIs to implement at the emulator, as shown in Table 2.1. In the pattern API (the API we expect most users to follow), users implement four functions, as illustrated in Figure 2-2. These functions divide packet processing into routing (choosing an output port), classification (choosing amongst multiple queues for a given port), enqueuing

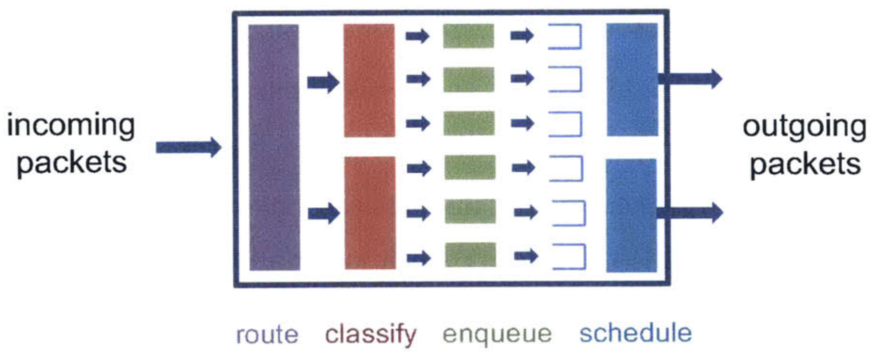

Figure 2-2: The four components of the emulator pattern API.

(adding a packet to the chosen queue or dropping it), and scheduling (choosing a packet to dequeue from a port). The framework calls the route, classify, and enqueue functions in order for each packet arriving at a router, and calls the schedule function on each output port in a router once per timeslot. Each of these functions is implemented in a separate C++ class, enabling them to be composed arbitrarily. For example, the same RED queue manager can be run in a router with one queue per port and FIFO scheduling, or with multiple queues fed by a flow classifier and scheduled using a deficit round-robin scheduler.

If the user wants more flexibility than the pattern API, we provide a more general API specified in the generic row of Figure 2.1. The generic API is for schemes that perform the functions above in a different order. For example, a scheme might enqueue packets to a shared queue and delay their routing decisions until an output port becomes available. The general API supports such a scheme; the framework simply calls input and output on each router once per timeslot. Input delivers a batch of arriving packets; output requests a batch of packets to transmit to the next hop. The general API also includes a function new which delivers a batch of packets from the network stack to a group of emulated endpoints.

Flexplane also provides a C API for users to obtain and modify packet headers at the physical endpoints. For schemes that require access to packet fields beyond the default abstract packet fields, the user specifies how to copy the desired custom information from the real packet into the abstract packet, by implementing the function `prepare_request`. Flexplane then conveys this data to the emulator along with the rest of the abstract packet so that it will be available to the emulated scheme.

Similarly, users may add support at the endpoints for modifying packet headers. To add modifications such as ECN marks, users implement the function `prepare_to_send`. This function is called for each packet immediately before it is sent on the real network and includes a pointer to the packet as well as an array of bytes from the abstract packet, populated by the resource management scheme in the emulator.

2.5 Scaling the Emulator with Multicore

Supporting large networks with Flexplane remains a significant challenge, primarily because it requires high aggregate emulator throughput. Our emulator design is motivated by the steady increase in the number of cores available on modern multicore machines. It is not uncommon today to build machines with multiple processors and dozens, if not hundreds, of cores. As of mid 2015, individual processors with between eight and 256 cores are commercially available on machines [1]. The emulator runs on a multicore machine; as long as each additional core increases total throughput, the emulator can scale to large networks by using many cores.

Networks of routers and endpoints are naturally distributed, allowing the emulated network to be easily distributed across many cores. Flexplane employs a simple strategy of assigning each router to a separate core. Typically a group of endpoints will also be handled by the core that handles their top of rack switch. With this assignment, cores only need to communicate when their routers have packets to send to or receive from a different core. The Flexplane framework handles this communication with FIFO queues of abstract packets between cores that have connected network components.

Flexplane employs three strategies to reduce the overhead of inter-core communication. First, Flexplane maintains only *loose synchronization* between cores. Each core independently ensures that it begins each timeslot at the correct time, but cores do not explicitly synchronize with each other. Tight synchronization, which we attempted with barriers, is far too inefficient to support high throughput. Second, Flexplane *batches* accesses to the FIFO queues; all abstract packets to be sent on or received from a queue in a single timeslot are enqueued or dequeued together. This is important for reducing contention on shared queues. Third, Flexplane *prefetches* abstract packets the first time they are accessed on a core.[3] This is possible because packets are processed in batches; later packets can be prefetched while the core performs operations on earlier packets.

To handle large routers and computationally heavy schemes, we propose to split a single router across multiple cores. One way to do this is to divide the router up by its output ports and to process different output ports on different cores; this approach makes sense because different output ports typically have little or no shared state.

2.6 Fault Tolerance

Flexplane provides reliable delivery of abstract packets both to and from the emulator. Because each abstract packet corresponds to a specific packet or group of packets (uniquely identified by their flow and sequential index within the flow), they are not interchangeable; for correct behavior, the emulator must receive an abstract packet for each real packet. If the physical network supports simple priority classes, then the traffic between endpoints and the emulator should run at the highest priority, making these packets unlikely to be dropped. Flexplane uses ACKs and timeouts to retransmit any abstract packets in the unlikely event that they are still dropped.

The emulator maintains only soft state so that another emulator can easily take over on emulator failure. We handle emulator fail-over in the same way as in Fastpass [37]: a secondary emulator takes over when it stops receiving periodic watchdog packets from the primary, and endpoints update the new emulator with their pending abstract packets.

[3]Previous work has shown that prefetching can significantly increase the instructions per cycle in packet-processing workloads [24].

2.7 Deployment

A typical datacenter consists of racks of machines connected to top-of-rack (ToR) switches, which are then connected redundantly to aggregate and core switches in higher tiers. Figure 2-1 shows a simple canonical example.

Flexplane provides a programmable data plane either for the entire network or for a contiguous sub-topology of the network. For ease of exposition (and because it is the expected deployment mode), the design presented here is for the case when the entire network is being handled.

We assume that servers in the network support and run any part of a scheme that requires an endpoint implementation (e.g., to run the endpoint's DCTCP or XCP software). The emulator emulates from the endpoint NICs onwards, and the emulated endpoints act simply as sources and sinks to the rest of the emulation.

Chapter 3

Implementation

We implemented Flexplane by extending Fastpass [37]. The Flexplane emulator is written in a mixture of C and C++ and uses the Intel Data Plane Development Kit (DPDK) [23] for low-latency access to NIC queues from userspace. Support for Flexplane at the endpoints is implemented in C in a Linux kernel module, which functions as a Linux `qdisc`, intercepting and enqueueing packets just before they are passed to the driver queue.

To program the network, users (typically network administrators) write modest amounts of code. They implement the router behavior in the emulator by implementing the pattern or generic C++ API shown in Table 2.1. With the pattern API, a router is instantiated with a routing table, a classifier, a queue manager, and a scheduler. Users can derive their own classes for these components from existing ones and adapt them for their needs. To add custom fields to abstract packets or modifications to packets, users also implement the endpoint C API.

We have implemented and experimented with several schemes in Flexplane: DropTail, RED (with and without ECN), DRR, priority queueing, DCTCP, and HULL. Most schemes require only a few dozen lines of code and the most complex (RED) requires 125 lines, as shown in Table 3.1. To show the simplicity of implementation, we show the key portions of the source code for DCTCP and priority scheduling in Figures 3-1 and 3-2, respectively.

scheme	LOC
DropTail Queue Manager	39
RED Queue Manager	125
DCTCP Queue Manager	43
Priority Queueing Scheduler	29
Round Robin Scheduler	38
HULL Scheduler	60

Table 3.1: Lines of code in the emulator for each resource management scheme.

```
struct dctcp_args {
    uint16_t q_capacity;
    uint32_t K_threshold;
};

class DCTCPQueueManager : public QueueManager {
public:
  void DCTCPQueueManager::enqueue(AbstractPkt *pkt,
    uint32_t port, uint32_t queue, uint64_t cur_time,
    Dropper *dropper) {
    uint32_t qlen = m_bank->occupancy(port, queue);
    if (qlen >= m_dctcp_params.q_capacity) {
      /* no space to enqueue, drop this packet */
      dropper->drop(pkt, port);
      return;
    }

    /* mark if queue occupancy is greater than K */
    if (qlen > m_dctcp_params.K_threshold) {
      /* Set ECN mark on packet */
      dropper->mark_ecn(pkt, port);
    }

    m_bank->enqueue(port, queue, pkt);
  }
private:
  PacketQueueBank *m_bank;
  struct dctcp_args m_dctcp_params;
}
```

Figure 3-1: Source code for a DCTCP queue manager in Flexplane

```
class Priority Scheduler : public Scheduler {
public:
  AbstractPkt* PriorityScheduler::schedule(uint32_t port,
    uint64_t cur_time, Dropper *dropper) {
    /* get the mask of non-empty queues for this port */
    uint64_t mask = m_bank->non_empty_qmask(port);

    uint64_t q_index;
    /* bsfq: find the first set bit of mask in 1 x86 instruction */
     asm("bsfq %1,%0":"=r"(q_index):"r"(mask));

     return m_bank->dequeue(port, q_index);
   }
private:
  PacketQueueBank *m_bank;
}
```

Figure 3-2: Source code for a priority scheduler in Flexplane over ≤ 64 queues.

Chapter 4

Evaluation

We conduct experiments on a single rack of 39 servers, each connected to a top-of-rack (ToR) switch. Each server has two CPUs, each with 20 logical cores, 32GB RAM, one 10 Gbits/s NIC, and runs the Linux 3.18 kernel. One server is set aside for running the emulator; we use 6WIND's PMD driver to run our DPDK-based emulator with a 10Gbits/s NIC. The switch is a Broadcom Trident+ based device with 64x10 Gbits/s ports and 9 MBytes of shared buffer. The switch supports a few schemes such as WRED with ECN marking, which we disable in all experiments except on an explicit comparison for emulation accuracy. We use an MTU size of 1500 bytes and disable TCP segmentation offload (TSO), a feature not yet supported in Flexplane.

Our experiments aim to answer the following questions:

1. Accuracy: How faithfully does Flexplane recreate the behavior of schemes already supported in hardware? How do packet latency and throughput compare to hardware?
2. Protocol experimentation: How can Flexplane be used to experiment with schemes not yet supported in hardware?
3. Emulator scalability: How does the throughput of the emulator scale with the addition of more cores?

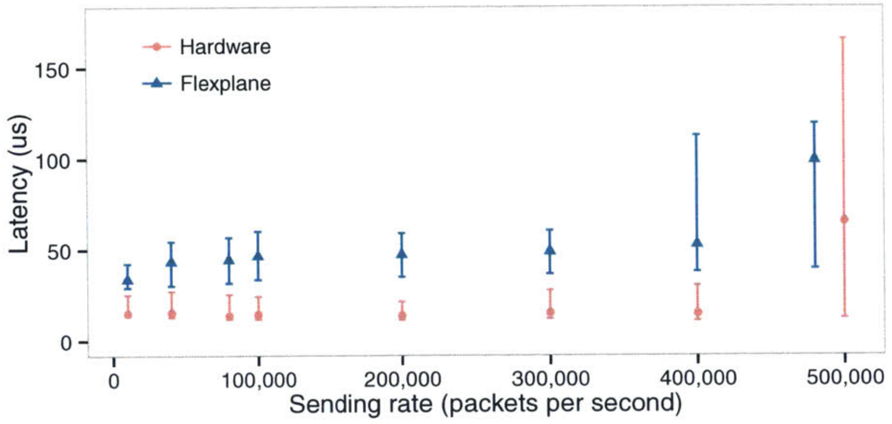

Figure 4-1: Packet latency under varying loads for one sender in Flexplane and the baseline hardware network. Points show medians; error bars show minimum and 99th percentile observed over 30 seconds.

4.1 Accuracy

Latency. First we compare the latency of a packet in Flexplane to that of a packet running on bare hardware, in an uncongested network. For this experiment, one client sends MTU-sized UDP packets to a server in the same rack, using the `sockperf` utility [2]; the receiver responds periodically to the sender at a rate of 10,000 responses per second. We measure the RTT of the response packets at several different rates of packets sent per second, and estimate the one-way latency as the RTT divided by two. We run DropTail both in Flexplane and on the hardware switch.

The results in Figure 4-1 demonstrate that the per-packet latency overhead of Flexplane is modest. Under the lightest offered load we measure (10,000 packets/s), the median latency in Flexplane is 33.8 µs, compared to 14.9 µs on hardware. As the load increases, the latency in Flexplane increases slightly due to the additional load on the kernel module in the sending endpoint. Flexplane is unable to meet the highest offered load (6 Gbits/s), because of the CPU overhead of the kernel module.

Throughput and fairness. Next we evaluate Flexplane's accuracy for bulk-transfer TCP, using application-level metrics: throughput and fairness across flows. In each experiment, five machines send TCP traffic at maximum throughput to one receiver for five minutes.

	Jain's Fairness Index			
	Hardware		Flexplane	
	median	σ	median	σ
DropTail	0.969	0.027	0.968	0.028
RED	0.987	0.011	0.980	0.011
DCTCP	0.996	0.004	0.990	0.006

Table 4.1: Flexplane achieves similar Jain's fairness index and standard deviation in fairness index (σ) as hardware. Medians and deviations are computed over 1-second intervals.

We compare Flexplane to hardware for three schemes that our router supports: TCP-cubic/DropTail, TCP-cubic/RED, and DCTCP. We configure the hardware router and the emulator using the same parameters for each scheme. For DropTail we use a static per-port buffer size of 1024 MTUs. For RED, we use min_th=150, max_th=300, max_p=0.1, and weight=5.[1] For DCTCP, we use an ECN-marking threshold of 65 MTUs, as recommended by its designers [6].

Flexplane achieves similar aggregate throughput as the hardware. All three schemes consistently saturate the bottleneck link, achieving an aggregate throughput of 9.4 Gbits/s in hardware, compared to 9.2-9.3 Gbits/s in Flexplane. This 1-2% difference in throughput is due to bandwidth allocated for abstract packets in Flexplane.

Flexplane also displays similar trends in fairness between flows as the hardware implementations. For each second of each experiment, we compute Jain's fairness index; Table 4.1 shows the median fairness index and the standard deviation of measured fairness indices. All schemes provide very high fairness, with DropTail providing slightly less and DCTCP slightly more fairness than RED; Flexplane demonstrates this behavior as well. Similarly, DropTail demonstrates the most variation in fairness over the course of the experiment and DCTCP the least.

Queueing. Next we evaluate the network-level behavior of Flexplane. During the experiment described above, we sample the total buffer occupancy in the hardware router every millisecond, and the emulator logs the occupancy of each emulated port

[1]This means the average queue is computed using an exponentially weighted moving average with weight $2^{-5} = \frac{1}{32}$.

	Queue Occupancies (MTUs)			
	Hardware		Flexplane	
	median	σ	median	σ
DropTail	931	73.7	837	98.6
RED	138	12.9	104	32.5
DCTCP	61	4.9	51	13.0

Table 4.2: Flexplane achieves similar queue occupancies and standard deviations in occupancies (σ) as hardware.

at the same frequency.

Table 4.2 shows that Flexplane maintains similar queue occupancies as the hardware schemes. For DropTail it maintains high occupancies (close to the max of 1024) with large variations in occupancy, while for the other two schemes the occupancies are lower and more consistent. Flexplane does differ from hardware in that its occupancies tend to be slightly lower and to display more variation. We believe this is due to the effectively longer RTT in Flexplane. When the congestion window is reduced, the pause before sending again is longer in Flexplane, allowing the queues to drain more. Thus, Flexplane effectively emulates a network with a longer RTT.

During the Flexplane experiments, the hardware queue sizes remain small: the mean is 7-10 MTUs and the 95th percentile is 14-22 MTUs. These numbers are small compared to the queue sizes in the emulator or in the hardware queues during the hardware experiments, and indicate that queueing in the hardware network does not significantly impact the accuracy of Flexplane (§2.3).

4.2 Experimenting with Flexplane

In this section, we provide an example of how one can use Flexplane to experiment with different resource management schemes. We do not argue that any scheme is better than any other, but instead demonstrate that there are tradeoffs between different schemes (as previously described in [44]), and that Flexplane can help users explore these tradeoffs.

We demonstrate the tradeoff that schemes make between performance for short

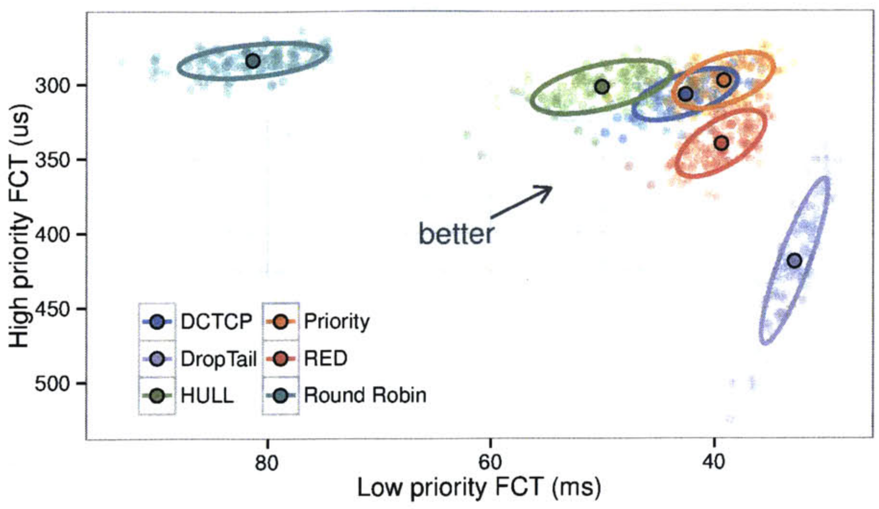

Figure 4-2: Flexplane enables users to explore tradeoffs between different schemes. Large points show averages over the entire experiment, faded points show averages over 1s, and ellipses show one standard deviation.

flows and performance for long flows. We use a simple RPC-based workload in which 4 clients repeatedly request data from 32 servers. 80% of the requests are short 1.5 KB "high priority" requests, while the remaining 20% are 10 MB "low priority" requests. Request times are chosen by a Poisson process such that the client NICs are receiving at about 60% of their maximum throughput. We run each scheme for five minutes.

We consider the schemes discussed above, as well as TCP-cubic/per-source-DRR, TCP-cubic/priority-queueing, and HULL.[2] For HULL, we use the recommended phantom queue drain rate of 95% of the link speed (9.5 Gbits/s). The HULL authors use a 1 KB marking threshold in a 1 Gbits/s network [5], and suggest a marking threshold of 3-5 KB for 10 Gbits/s links. However, we find that throughput degrades significantly with a 3 KB marking threshold, achieving only 5 Gbits/s total with 4 concurrent flows. We increase the marking threshold until our achieved throughput is 92% of the drain rate (this is what [5] achieves with their parameters); the resulting threshold is 15 KB.

Figure 4-2 shows the tradeoff each scheme makes on the RPC workload. With DropTail, large queues build up in the network, leading to high flow completion times

[2]We omit the pacer at the endpoints, using only the phantom queues.

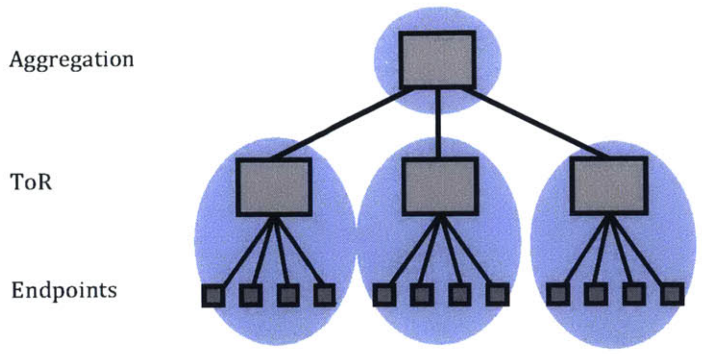

Figure 4-3: One of the datacenter network topologies used to generate the scalability results shown in Figure 4-4. Each rack has 32 endpoints; only a subset are depicted for simplicity. The purple ovals indicate network components handled by the same CPU core.

(FCTs) for the high priority requests. However, DropTail senders rarely cut back their sending rates and therefore achieve the best FCTs for the long requests. At the other end of the spectrum, round robin's small static buffers (32 MTUs per queue) cause the long flows to drastically cut back their sending rates, allowing the short flows to achieve low FCTs. The other schemes make trade-offs between the two. HULL's phantom queues cause senders to cut back their sending rates more aggressively than DCTCP, leading to slightly better FCTs for short flows but worse FCTs for long flows, compared to DCTCP.

4.3 Emulator Scalability

In this section, we seek to demonstrate that the Flexplane emulator has the potential to support large networks. We cannot yet demonstrate that the emulator can support networks with hundreds or thousands of endpoints; in fact, we have not yet tested it on such large networks. However, our experiments demonstrate that, for small numbers of cores, supplying additional cores to the emulator enables it to sustain higher aggregate throughput. This suggests that the design could scale; we defer full scalability to future work.

Scaling with number of cores. We consider several variants on a simple datacenter network topology, each of a different size. Each topology includes racks of 32 endpoints connected via a ToR switch. Multiple ToRs are then connected using a single aggregation switch. We assign the aggregation (Agg) switch to its own CPU core, and then assign each ToR and its adjacent endpoints to another core, so that the number of emulation cores used for a topology is one plus the number of racks. An example topology with three racks is shown in Figure 4-3; the ovals indicate which network components are run on the same CPU core.

We generate a synthetic workload using two additional cores on the same machine. As we vary the number of racks in the topology, we also vary the fraction of traffic whose source and destination are in different racks so that all routers (ToR and Agg) always process the same amount of network throughput. Sources and destinations are chosen uniformly at random with this constraint. Timings obey Poisson arrivals and we vary the mean inter-arrival time to produce different network loads.

We run an automated stress test to determine the maximum sustainable throughput for a given configuration. It begins with low load and increases the load every 5 seconds as long as all cores are able to sustain it. Whenever one core falls behind, processing timeslots later than their scheduled time, the stress test decreases the network load. This process of repeatedly adjusting the applied load continues for 2 minutes. It then reports the total throughput observed over the last 30 seconds. For each configuration, we run the stress test five times and average the results. We run this experiment on a large multicore machine with 8 CPUs, each with 10 cores and 32GB of RAM. Each CPU is a 2.4 GHz Intel Xeon CPU E7-8870.

Figure 4-4 shows the maximum throughput achieved as we vary the number of racks in the topology from two to seven, and correspondingly the number of emulation cores from three to eight. We measure throughput in two ways; endpoint throughput is the amount of traffic sent/received by the endpoints and router throughput is the total amount of traffic processed by all routers in the network.

Router throughput increases approximately linearly with the number of emulation cores. In the configuration with 2 racks and 3 cores, the emulator achieves about 244 Gbits/s of throughput. Each additional core provides 22 Gbits/s of additional throughput on average until the final configuration with seven racks achieves 352

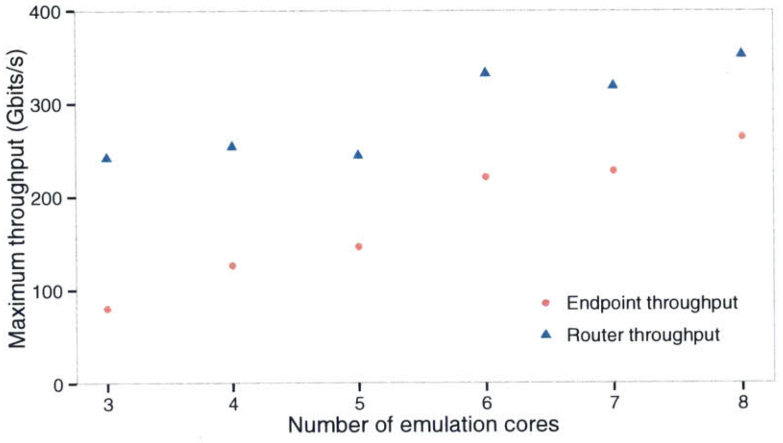

Figure 4-4: Maximum throughput achieved by the Flexplane emulator for different numbers of emulation cores. Endpoint throughput is the throughput sent/received by the endpoints; router throughput is the total throughput sent/received by all routers in the network.

Gbits/s. This makes sense; router throughput should scale linearly with the number of cores because the workload is chosen such that all routers (and therefore all cores) receive the same amount of network traffic. Endpoint throughput remains lower than router throughput, but approaches router throughput as the size of the topology increases.

These results suggest that the 80 core machine used for these experiments could support about 1.7 Terabits/s of router throughput. Each stress test core can supply about 200-225 Gbits/s of endpoint throughput during a stress test; when running with live traffic we expect each communication core to support similar throughput or slightly less. Thus about 9 of the cores could be used for communicating with endpoints in the network. The first 3 of the remaining cores would supply 244 Gbits/s of throughput and the remaining 68 would supply about 22 Gbits/s each, for 1.7 Terabits/s of router throughput total. Our results presented here only use cores on the same socket. Scaling across sockets presents additional challenges because the latency of communication between cores increases; we leave this for future work.

Comparison with previous work. Here we consider how Flexplane compares to existing software routers. Flexplane outperforms existing approaches in terms of individual router capacity. When emulating only a single router with 32 endpoints, Flexplane achieves a total throughput of about 90 Gbits/s, compared to a maximum of 35 Gbits/s in RouteBricks [14] and 40 Gbits/s in PacketShader [19], with minimal forwarding.

The difference between Flexplane and previous approaches is even more pronounced when we compare throughput per clock cycle, to normalize for different numbers of cores used and different clock frequencies. In its largest configuration, RouteBricks achieved 35 Gbits/s of throughput with 32 2.8 GHz cores. In comparison, our 7 rack configuration achieved a total router throughput of 352 Gbits/s with 10 2.4 GHz cores total (8 emulation cores plus 2 stress test or communication cores). This amounts to 38 times as much throughput per clock cycle in Flexplane. The difference arises because Flexplane performs no processing on its forwarding path involving data or header manipulation, leaving the hardware routers to handle that, and focusing only on resource management functions.

Chapter 5

Related Work

Existing approaches to designing a programmable data plane fall into two broad categories: software routers and programmable hardware.

5.1 Software routers

Software routers process packets using general-purpose processors rather than hardware routers. These approaches are very flexible and can scale to support large networks, but to date have demonstrated very limited router capacity. The best results today typically demonstrate 30-40 Gbits/s of router capacity; this is only 10% of the capacity needed to support a small router with 32 links at today's datacenter speeds (10 Gbits/s).

Click [26] is a popular modular software router built as a pipeline of extensible elements. It has been extended to take advantage of multiprocessor machines [10]. In addition, RouteBricks shows how to deploy Click across multiple servers; with a 4-server configuration, each with 8 2.8 GHz cores, RouteBricks achieves an aggregate throughput of 35 Gbits/s [14]. A key idea in RouteBricks is the use of Valiant load balancing to switch traffic. In contrast, Flexplane incurs no such switching overhead because the packet payloads do not pass through the emulator. Flexplane requires much less CPU and network bandwidth than RouteBricks to achieve the same router throughput, because it processes short abstract packets instead of entire packet payloads.

Other approaches such as Packet Shader [19] and GSwitch [50] use GPUs to accelerate packet-processing in software; Packet Shader achieves 39 Gbits/s on a single commodity PC. However, invoking a GPU increases processing time significantly, by around 30 µs in modern hardware [24].

ServerSwitch [30] enables users to program customized packet forwarding in a commodity Ethernet switching chip, but all other processing is still done on a general-purpose CPU. The authors demonstrate an implementation of the QCN [52] (Quantized Congestion Notification) scheme to show the viability of this approach. The design of ServerSwitch, however, does not accommodate the variety of network resource management schemes supported by Flexplane at high rates, because in order to achieve the desired flexibility, most packets must traverse the software router running on the general-purpose CPU.

Sidecar [42] is a position paper proposal that distributes processing code between endpoints and general-purpose processors connected to each switch via standard redirection mechanisms. Packets are diverted toward these processors from switches when they need custom software to run over them. Sidecar has not been built, but it is not clear how the design can run at near-hardware rates; the assumption is that only a small fraction of packets will require software processing.

NetBump [4] is a platform for AQM experimentation; it acts as a "bump on the wire," supporting line rate AQM and forwarding. Netbump is implemented in software as a zerocopy, kernel-bypass network API and demonstrates excellent line-rate performance (14.17 million packets per second supporting a 10 Gbits/s link). Unlike Netbump, Flexplane seeks to support a wider range of schemes, and is deployed off-path rather than by augmenting each switch. As such, it may be easier to program, and may be cheaper as the network size grows.

Other works focus on techniques to increase packet processing performance. Many software routers face a "memory wall" in processing packets [36]; this paper argued that data caches and multi-threading form the minimal set of hardware mechanisms that achieve both programmability and performance. In subsequent work, they used this observation to develop a "malleable" processor architecture to reconfigure cache capacity and the number of threads [35].

Flexplane relies on fast, kernel-bypassed, zerocopy network access for the emulator. Multiple frameworks such as Intel's DPDK [23], netmap [38], and PF_RING [13] enable such low-latency packet processing; Flexplane uses DPDK.

The approach used in Flexplane builds on the idea of centralized per-packet arbitration developed in Fastpass [37]. Unlike Fastpass, this paper shows how to emulate a variety of network resource management schemes, not just a particular zero-queue objective.

5.2 Programmable hardware

An alternative to programming most or all packet processing in software is to augment hardware components with some programmable APIs. These approaches typically run at hardware rates and can scale to large networks, but have limited flexibility.

Many of these approaches are based on FPGAs. An early project, P4 [18] uses FPGAs to build an active network switch. Other FPGA-based designs were later proposed by multiple groups working on dynamic hardware plugins [47, 27, 22]. The DHP [47] platform, for example, processes flows in both hardware and software.

NetFPGA [29] is a popular open source hardware forwarding platform that uses an FPGA-based approach and provides some flexibility. RiceNIC [41] has a similar design, but adds the ability to program via two general-purpose processors. Switchblade [8] is a NetFPGA-based pipelined system for router programmability with software exceptions invoked using packet or flow rules; it is designed primarily for header and protocol manipulation, not resource management functions. Chimpp [39] is a modular router design also built atop NetFPGA; it aims to remove restrictions on module interfaces and programming style. It supports Click-like configurations, like earlier work on Cliff [28] and CUSP [40]. CAFE [31] uses NetFPGA to provide configurability in datacenter networks; its API allows a user to configure packet forwarding, and modify, insert, and delete any header fields without changing hardware.

In our view, CAFE is designed for header field manipulations, but less so for network resource management.

A recent position paper [44] proposes a way of extending software-defined networking to the data plane using programmable hardware elements. It uses FPGAs and focuses on network resource management. The motivation for that work is the same as for Flexplane, but our approach is to program schemes in high-level software on commodity hardware.

Despite the abundance of work on FPGA-based platforms, FPGAs are not an ideal platform for datacenter development. FPGAs are not a comfortable or convenient programming platform when compared to programming in C++ as in Flexplane, and cannot offer the same flexibility. It is also worth noting that FPGAs are rarely used today (if at all) in production network equipment; our goal with Flexplane is to provide a way to program production network infrastructure.

The P4 language [9] is at the forefront of recent work on flexible header manipulation. It aims to provide a standard header manipulation language to be used by SDN control protocols like OpenFlow [34], with commercial switches developing hardware to process programs expressed in P4. As a domain-specific language, P4 improves usability but not flexibility, and still falls short of the usability of C++. In addition, P4 does not yet provide the primitives necessary to support most resource management schemes.

Chapter 6

Conclusion and Future Work

Flexplane is a new approach to programmable resource management in datacenter networks. Our prototype implementation demonstrates accuracy, ease of experimentation, and throughput that increases linearly for small numbers of cores. However, there are several remaining aspects of Flexplane that we have yet to address in full:

- Large networks: How can we scale the emulator to support the aggregate throughput of a large network? How can we provide high capacity for large emulated routers? How can Flexplane cope with different latencies between endpoints and the emulator?
- Fault tolerance: How can Flexplane handle failures of network components (e.g. links, routers, etc.)?
- Endpoint overheads: How can we reduce the CPU overhead at the endpoints or support TCP segmentation offload so that a Flexplane endpoint can achieve full throughput with a single core?

If these questions can be addressed satisfactorily, Flexplane has the potential to increase programmability in datacenters, benefiting researchers, network operators, and students in networking courses.

Bibliography

[1] http://en.wikipedia.org/wiki/Multi-core_processor.

[2] https://github.com/mellanox/sockperf.

[3] Matthew Adiletta, Mark Rosenbluth, Debra Bernstein, Gilbert Wolrich, and Hugh Wilkinson. The next generation of Intel IXP network processors. *Intel technology journal*, 6(3):6–18, 2002.

[4] Mohammad Al-Fares, Rishi Kapoor, George Porter, Sambit Das, Hakim Weatherspoon, Balaji Prabhakar, and Amin Vahdat. Netbump: user-extensible active queue management with bumps on the wire. In *ANCS*, 2012.

[5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *NSDI*, 2012.

[6] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[7] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 435–446. ACM, 2013.

[8] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. Switchblade: A platform for rapid deployment of network protocols on programmable hardware. In *SIGCOMM*, 2011.

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[10] Benjie Chen and Robert Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *USENIX Annual Technical Conference*, 2001.

[11] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: A software architecture for next generation routers. In *SIGCOMM*, 1998.

[12] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulations of a Fair-Queueing Algorithm. *Internetworking: Research and Experience*, V(17):3–26, 1990.

[13] Luca Deri et al. Improving passive packet capture: Beyond device polling. In *Proceedings of SANE*, volume 2004, pages 85–93, 2004.

[14] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *SOSP*, 2009.

[15] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[16] S. Floyd. TCP and Explicit Congestion Notification. *CCR*, 24(5), October 1994.

[17] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Trans. on Networking*, 1(4), August 1993.

[18] Ilija Hadžić and Jonathan M Smith. P4: A platform for FPGA implementation of protocol boosters. In *Field-Programmable Logic and Applications*, 1997.

[19] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. In *SIGCOMM*, 2011.

[20] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell, and JB Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 2008.

[21] C. Y. Hong, M. Caesar, and P. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, 2012.

[22] Edson L Horta, John W Lockwood, David E Taylor, and David Parlour. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *Proceedings of the 39th annual Design Automation Conference*, 2002.

[23] Packet processing is enhanced with software from intel DPDK. `http://www.intel.com/go/dpdk`.

[24] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G Andersen. Raising the bar for using gpus in software packet processing. In *Proc. NSDI*, 2015.

[25] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[26] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[27] Fred Kuhns, John Dehart, Anshul Kantawala, Ralph Keller, John Lockwood, Prashanth Pappu, David Richard, David Taylor, Jyoti Parwatikar, Ed Spitznagel, et al. Design and evaluation of a high-performance dynamically extensible router. In *DARPA Active NEtworks Conference and Exposition (DANCE)*, 2002.

[28] Chidamber Kulkarni, Gordon Brebner, and Graham Schelle. Mapping a domain specific language to a platform FPGA. In *ACM Design Automation Conference*, 2004.

[29] John W Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *IEEE International Conf. on Microelectronic Systems Education*, 2007.

[30] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. Serverswitch: A programmable and high performance platform for data center networks. In *NSDI*, 2011.

[31] Guohan Lu, Yunfeng Shi, Chuanxiong Guo, and Yongguang Zhang. CAFE: a configurable packet forwarding engine for data center networks. In *ACM SIGCOMM workshop on programmable routers for extensible services of tomorrow*, 2009.

[32] Steven McCanne, Sally Floyd, and Kevin Fall. The lbnl network simulator. *Software on-line: http://www.isi.edu/nsnam*, 1997.

[33] Paul E McKenney. Stochastic fairness queueing. In *INFOCOM*, 1990.

[34] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[35] Jayaram Mudigonda, Harrick M Vin, and Stephen W Keckler. Reconciling performance and programmability in networking systems. In *SIGCOMM*, 2007.

[36] Jayaram Mudigonda, Harrick M Vin, and Raj Yavatkar. Overcoming the memory wall in packet processing: hammers or ladders? In *ANCS*, 2005.

[37] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Deverat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*, 2014.

[38] Luigi Rizzo. netmap: a novel framework for fast packet i/o. In *USENIX ATC*, 2012.

[39] Erik Rubow, Rick McGeer, Jeff Mogul, and Amin Vahdat. Chimpp: A Click-based programming and simulation environment for reconfigurable networking hardware. In *ANCS*, 2010.

[40] Graham Schelle and Dirk Grunwald. CUSP: a modular framework for high speed network applications on FPGAs. In *ACM/SIGDA Symposium on Field-programmable Gate Arrays*, 2005.

[41] Jeffrey Shafer and Scott Rixner. Ricenic: A reconfigurable network interface for experimental research and education. In *Proceedings of the 2007 workshop on experimental computer science*, 2007.

[42] Alan Shieh, Srikanth Kandula, and Emin Gun Sirer. Sidecar: building programmable datacenter networks without programmable switches. In *HotNets*, 2010.

[43] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996.

[44] Anirudh Sivaraman, Keith Winstein, Suvinay Subramanian, and Hari Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *HotNets*, 2013.

[45] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. In *SOSP*, 2001.

[46] C.H. Tai, J. Zhu, and N. Dukkipati. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM*, 2008.

[47] David E Taylor, Jonathan S Turner, John W Lockwood, and Edson L Horta. Dynamic hardware plugins: exploiting reconfigurable hardware for high-performance programmable routers. *Computer Networks*, 38(3):295–310, 2002.

[48] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[49] András Varga et al. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, volume 9, page 185. sn, 2001.

[50] Matteo Varvello, Rafael Laufer, Feixiong Zhang, and TV Lakshman. Multi-layer packet classification with graphics processing units. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 109–120. ACM, 2014.

[51] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 50–61. ACM, 2011.

[52] Yan Zhang and Nirwan Ansari. On mitigating tcp incast in data center networks. In *INFOCOM*, 2011.