Flexplane: An Experimentation Platform for Resource Management in Datacenters

Amy Ousterhout, Jonathan Perry, and Hari Balakrishnan, *MIT CSAIL;*
Petr Lapukhov, *Facebook*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ousterhout

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Flexplane: An Experimentation Platform for Resource Management in Datacenters

Amy Ousterhout, Jonathan Perry, Hari Balakrishnan (MIT CSAIL), Petr Lapukhov (Facebook)

Abstract

Flexplane enables users to program data plane algorithms and conduct experiments that run real application traffic over them at hardware line rates. Flexplane explores an intermediate point in the design space between past work on software routers and emerging work on programmable hardware chipsets. Like software routers, Flexplane enables users to express resource management schemes in a high-level language (C++), but unlike software routers, Flexplane runs at close to hardware line rates. To achieve these two goals, a centralized emulator faithfully *emulates*, in real-time on a multi-core machine, the desired data plane algorithms with very succinct representations of the original packets. Real packets traverse the network when notified by the emulator, sharing the same fate and relative delays as their emulated counterparts.

Flexplane accurately predicts the behavior of several network schemes such as RED and DCTCP, sustains aggregate throughput of up to 760 Gbits/s on a 10-core machine (≈ 20× faster than software routers), and enables experiments with real-world operating systems and applications (e.g., Spark) running on diverse network schemes at line rate, including those such as HULL and pFabric that are not available in hardware today.

1 Introduction

Recently the networking community has witnessed a renewed flurry of activity in the area of programmability, with the goal of enabling experimentation and innovation [17, 25, 34, 38]. Programmable networks facilitate experimentation at several different levels: researchers can experiment with new network protocols, network operators can test out new protocols before deployment, developers can debug applications with new network optimizations, and students can implement, run, and measure network algorithms on real hardware.

Much of this work has focused on the control plane; platforms for programming resource management algorithms in the data plane remain limited. Examples of such algorithms include:

- *Queue management*: what packet drop or ECN-marking [23] policy should be used? Examples include RED [24], DCTCP [10], HULL [11], and D^{2}TCP [50].
- *Scheduling*: which queue's packet should be sent next on a link? Examples include weighted fair queueing (WFQ) [20], stochastic FQ [37], priority queueing, deficit round-robin (DRR) [44], hierarchical FQ [16], pFabric [12], and LSTF [39].
- *Explicit control*: how should dynamic state in packet headers be created, used, and modified? Examples include XCP [32], RCP [22, 49], PDQ [29], D^3 [53], and PERC [30].

Because datacenters are typically controlled by a single entity and involve workloads with measurable objectives, they are an ideal environment for experimentation. Nevertheless, most proposed resource management schemes have been evaluated only in simulation because no existing platform enables experimentation in router data planes with the desired level of programmability and usability. Unfortunately *simulations capture only part of the story*; they don't accurately model the nuances of real network stacks, NICs, and applications.

Previous work has explored two approaches to enabling data plane programmability. First, programmable hardware [7, 13, 34, 45, 47] enables users to program some data plane functions such as header manipulations. To date programmable switching chips have not provided sufficient flexibility to support many resource management schemes. This remains an active area of research [45, 46], but is not a viable option today. Furthermore, even if programmable hardware proves fruitful, users must replace their existing hardware to take advantage of it. Second, software routers [19, 21, 27, 33, 48] offer excellent flexibility, but provide insufficient router capacity for many applications, with the best results providing an aggregate throughput of only 30-40 Gbits/s.

In this paper we explore an alternative approach to programming resource management schemes in switches. We develop a system called **Flexplane**, which functions as an intermediate point between software routers and hardware. It remains as programmable as software routers, but provides substantially higher throughput. Flexplane does not match the performance of hardware, but is still well-suited for prototyping by researchers and students, evaluating new networking schemes with real-world operating systems and applications, and small-scale production.

The key idea in Flexplane is to move the software that implements the programmable data plane "off-path". We implement a user-specified scheme in a *centralized emulator*, which performs *whole-network emulation* in real time. The emulator maintains a model of the network topology in software, and users implement their schemes in the emulated routers. Before each packet is sent on the real network, a corresponding emulated packet traverses the emulated network and experiences behavior specified by the resource management scheme. Once this is complete, the real packet traverses the network without delay. Thus, applications observe a network that supports the programmed scheme (Figure 1).

The rationale for this design is that resource management schemes are *data-independent*; they depend only on a small number of header fields, which typically comprise less than 1% of the total packet size.[1] Thus it suffices for resource management schemes to operate over short descriptions of packets, called *abstract packets*, rather than entire packets. This allows a software entity to support much higher aggregate throughput with the same network bandwidth.

We have implemented Flexplane, and have written seven different resource management schemes in it. We performed experiments to answer the question: *does off-path emulation over abstract packets provide a viable platform for experimentation with network resource management schemes?* Our results show that Flexplane provides accuracy, utility, and sufficient throughput for datacenter applications:

- **Accuracy:** Flexplane accurately reproduces the queue occupancies and flow completion times of schemes already supported in commodity hardware such as DropTail, RED, and DCTCP. For example, Flexplane matches the average normalized flow completion times for small flows in DCTCP to within 12% (§5.1).
- **Utility:** With Flexplane, users can implement a large number of schemes such as HULL, pFabric, etc. in a few dozen lines of code. They can use Flexplane to evaluate trade-offs between resource management schemes and to quickly tune protocol parameters for different link rates. Finally, users can experiment with real applications such as Spark and observe results that are not possible to observe in simulation, because they depend on the CPUs and network stacks of real endpoints (§5.2).
- **Throughput:** By limiting communication between cores, the Flexplane emulator scales nearly linearly to achieve 760 Gbits/s of throughput with 10 cores, for a topology of seven racks. This is about 20× as much as the RouteBricks software router while using one-third as many cores (§5.3).

2 Use Cases

In this section, we present three settings in which Flexplane can be useful and then describe the benefits of Flexplane over existing approaches.

Prototyping schemes. Network researchers frequently develop new resource management schemes. While prototyping, they need to test their schemes under realistic network conditions and to quickly iterate on their ideas.

Evaluating schemes. Many people are eager to try out new schemes and to evaluate the performance impact of them on their applications. For example, network operators have a choice of what switch algorithms to run in their networks and may consider upgrading their hardware to support schemes that are newly available in ASICs (e.g., CONGA [9]). Before replacing their hardware, however, operators must test a new scheme with their specific applications to evaluate the potential benefits. As another example, students may want to implement, run, and measure resource management schemes to better understand them.

Programmable resource management in production. Although implementing resource management schemes in hardware provides the best performance, Flexplane can be used to achieve custom resource management in a small production network in the interim until a new scheme is supported in hardware. Fastpass [40] can be used for a similar purpose but today supports only one objective (max-min fairness). In contrast, Flexplane provides flexible APIs that enable users to express arbitrary new policies in only dozens of lines of code. For example, if one wishes to run applications on HULL in a production setting, Flexplane provides that ability over existing hardware.

2.1 Benefits over Existing Approaches

Simulation. Today, many resource management schemes (e.g., PDQ [29], pFabric [12], PERC [30], and LSTF [39]) are developed and tested exclusively using network simulators such as ns [6, 28] and OMNeT++ [51]. Flexplane provides flexibility similar to that of simluators, and also provides three additional benefits.

First, simulations are insufficient by themselves to demonstrate a scheme's effectiveness, because they do not capture the nuances of real network stacks and NICs. For many resource management schemes, implementation details matter, and endpoint optimizations can inform protocol design. For example, burstiness due to Interrupt Coalescing and other features led the authors of HULL [11] to add an endpoint pacer to their design. These NIC features are not easy to model in simulations, and without testing in a real network, it is hard to predict which optimizations matter. By using real machines as endpoints, Flexplane inherently captures these behaviors, allowing researchers to better understand how their protocols would work in practice.

Second, Flexplane enables users to evaluate resource management schemes using real applications. Realistic applications are not solely network bound; they involve a complex mixture of network I/O, disk I/O, and computation. These aspects of applications are hard to model in network simulations, but can be captured naturally by running Flexplane with unmodified applications. Further-

[1] This contrasts with some router functions, such as encapsulation or encryption, which require the entire packet.

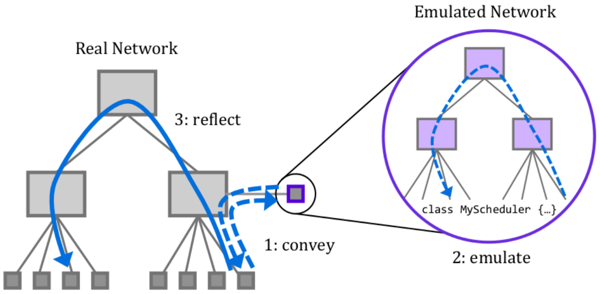


Figure 1: In Flexplane, endpoints *convey* their demands to the emulator in abstract packets, the emulator *emulates* packets traversing the network and experiencing the user-implemented scheme, and the results are *reflected* back onto the physical network. In the real network, dashed arrows represent an abstract packet, and the solid arrow represents the corresponding real packet.

more, with Flexplane users can evaluate real applications without porting them to simulation, which could be onerous for complex distributed systems (e.g., memcache or Spark). By enabling users to evaluate resource management schemes on real network hardware with real applications, Flexplane could mitigate risk for network operators and potentially enable greater adoption of new protocols.

Third, Flexplane experiments run in real time, completing much more quickly than simulations. For example, a 30ms ns-2 simulation with 100,000 flows took over 3 minutes to complete; this is 6000× slower than real time.

Software routers. Software routers are often used for research and experimentation (e.g., as in [54]), but Flexplane is a better approach for the use cases described above. Conducting large-scale experiments with software routers is infeasible because they provide insufficient throughput. Throughputs of 30-40 Gbits/s per router limit experiments to only a few servers per router with 10 Gbits/s links; Flexplane provides 20× as much throughput.

Programmable hardware. FPGAs and programmable switching chips do not provide the convenience of expressing schemes in C++, as in Flexplane, and do not enable experimentation in existing network infrastructure.

3 Design

Flexplane's goal is to enable a network to support a resource management scheme, even when the switch hardware does not provide it. 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 software in a single centralized multi-core server called the emulator, which endpoints consult before sending each packet.[2] The transmission of every packet in Flexplane involves three steps (Figure 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 (§3.1).
2. *Emulate:* The emulator models the entire network and emulates its behavior in real time. It delays and modifies abstract packets in the same way the corresponding real packets would be delayed and modified, if they traversed a hardware network implementing the same scheme (§3.2).
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 (§3.3).

These steps run in real time, with the result 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. Higher-layer protocols perceive that the packet is queued and modified by the resource management scheme implemented in the emulator. At the highest layer, datacenter applications experience a network that supports the emulated scheme.

3.1 Abstract Packets

An abstract packet concisely describes a chunk of data (e.g., a packet) to be sent onto the network. It includes the metadata required to route the packet through the emulated topology and the header fields that are accessed by routers running the chosen resource management scheme. All abstract packets include the source and destination addresses of the chunk of data, a unique identifier, and a flow identifier.

In addition, a scheme can include custom header fields in abstract packets, such as the type-of-service (DSCP) or whether the sender is ECN-capable. Flexplane provides a general framework that allows schemes to convey arbitrary packet information in abstract packets (§3.5). The ability to convey any information enables Flexplane to support existing resource management schemes as well as those that will be developed in the future. It also enables Flexplane to support schemes that require non-standard packet formats (e.g., pFabric [12]).

[2] Fastpass [40] previously introduced the idea of scheduling in a centralized entity at the granularity of individual packets. However, Fastpass schedules packets to achieve an explicit objective whereas Flexplane schedules packets to achieve the behavior of a distributed resource management scheme.

For simplicity of emulation, all abstract packets represent the same amount of data; this can consist of one large packet or multiple smaller packets in the same flow. In this paper, an abstract packet represents one maximum transmission unit (MTU) worth of packets (1500 bytes in our network).

Network bandwidth. Because communication between endpoints and the emulator occurs over the same network used to transmit real packets, it reduces the available network bandwidth (§6). However, abstract packets summarize only the essential properties of a chunk of data, and are quite small compared to the amount of data they represent. In a typical case, an abstract packet contains 2 bytes for each of its source, destination, flow, and unique identifier, and 4 bytes of custom data. These 12 bytes are less than 1% of the size of the corresponding MTU.

Abstract packets must be sent between endpoints and the emulator in real packets, adding additional overhead. However, multiple abstract packets can often be sent in the same real packet, e.g., when a single send() or sendto() call on a socket contains several packets worth of data. Abstract packets are also efficiently encoded into real packets; when an endpoint requests several packets in the same flow, the flow information is not included multiple times in the same packet to the emulator.

3.2 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 and with the same routing policies 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., ECN marks), or be dropped, just as they would in a real network. The emulator is, effectively, a real-time simulator.

To simplify emulation, the emulator divides time into short timeslots. 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. Thus 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.[3] Timeslots simplify the task of ensuring that events in the emulation occur at the correct time. Instead of scheduling each individual event (e.g., packet transmission) at a specific time, the emulator only needs to ensure that each timeslot (which includes dozens of events) occurs at the right time. This contrasts with ns [28, 6] and other event-based simulators.

We assume that servers in the network support and run any part of a scheme that requires an endpoint implementation (e.g., the endpoint's DCTCP or XCP software). This frees the emulator from the burden of performing computation that can be performed in software at the endpoints instead. The emulator only emulates in-network functionality, from the endpoint NICs onwards, and the emulated endpoints act simply as sources and sinks to the rest of the emulation.

3.3 Reflection

As soon as an abstract packet exits the emulation, the emulator sends a response to the source endpoint indicating how to handle the corresponding real packet on the real network. If the abstract packet was dropped during emulation, the emulator will instruct the real endpoint to drop the corresponding packet without sending it. Alternatively, if the abstract packet reached its emulated destination, the emulator will instruct the real endpoint to immediately modify the corresponding real packet (if necessary) and send it.

Modifying packets before sending them is required for correctness for schemes whose behavior depends upon received headers. For example, in ECN-based schemes [10, 11, 24, 50], routers set the ECN field in IP headers when congestion is experienced and the recipient must echo this congestion indication back to the sender. The receivers in explicit control schemes [22, 29, 30, 32, 53] similarly echo information about the network back to the senders.

Once a real packet has been modified, it will be sent onto the network. It will traverse the same path through the network that the corresponding abstract packet took through the emulation, because the emulator is configured to match the routing policies of the real network. The packet will not acquire more modifications in the real network, so it will arrive at its destination with the headers specified by the emulator.

3.4 Accuracy

To be a viable platform for experimentation, Flexplane must be accurate, meaning that its behavior predicts that of a hardware network. More precisely, we define *accuracy* as the similarity between results obtained by running an application over Flexplane with a given resource management scheme and results obtained by running the same application over a hardware network running the scheme.

Accuracy can be measured by comparing metrics at several different levels of the network stack. In section §5.1

[3] We assume, for simplicity, that all links in the network run at the same rate. If this is not the case, multiple abstract packets could be sent and received per timeslot on the higher-bandwidth links.

Emulator 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 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
Endpoints	prepare_request(sk_buff *skb, char *request_data)	copy abstract packet data into request_data
	prepare_to_send(sk_buff *skb, char *allocation_data)	modify packet headers before sending

Table 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.

we evaluate the extent to which Flexplane provides accuracy at the network and application levels, by analyzing metrics such as queue occupancies and flow completion time. Here we analyze Flexplane's accuracy at the granularity of individual packets, to better understand the ways in which Flexplane deviates from perfect accuracy.

To understand Flexplane's accuracy at the packet level, we compare the latency l' of a packet in a Flexplane network to the latency l of a packet in an identical hardware network that implements the same resource management scheme. For now, we assume that both networks consist of a single rack of servers. In the hardware 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 transmission delay (the ratio of the packet size to the link rate).

In the Flexplane network, the latency l' of a packet 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 u, and any queueing delay it encounters in the real network, q'. For an emulator in the same rack, $r \leq 2u$ and the emulation ensures that $t_e < u$.

$$l' = r + t_e + q_e + u + q' < 4u + q' + q_e \tag{1}$$

Flexplane does not guarantee zero queueing within the hardware network so q' may be nonzero, adding some jitter to l'. However, the emulation enforces bandwidth constraints, guaranteeing that there will be sufficient bandwidth to transmit all real packets admitted into the network to their destinations. This means that though there might be small amounts of transient queueing in the network, q' cannot grow indefinitely. In practice we find that q' is very small, approximately 8-12 µs on average for a fully utilized link (§5.1).

Emulation also contributes an additive delay to l'. Equation 1 estimates that for an unloaded network, the added latency in Flexplane is about three times the latency experienced in an equivalent hardware network, or less. However, our experiments indicate that in practice this additional latency is much smaller, about 1.3 times (§5.1). This difference is because a significant fraction of the unloaded delay u is spent in endpoint network stacks. The emulator uses kernel bypass, reducing r significantly below $2u$, and the emulation does not model endpoint network stacks, causing t_e to be significantly less than u. In addition, for loaded networks with congestion and queueing, these latency overheads contribute a smaller fraction of the overall latency, and Flexplane's latencies better match hardware.

Multiple racks. Flexplane supports multi-rack topologies. With multiple racks, the entire network is still scheduled using a single centralized emulator. As a result, round-trip times to the emulator and back (r) may vary slightly across endpoints. In addition, unloaded delays in the real network (u) will vary across packets that take different-length paths. To ensure that all packets experience the same delay overhead from Flexplane, endpoints delay packets so that the sum of r, u, and this additional delay equals the sum of the maximum values of r and u for any packet in the network. Thus with multiple racks, the delay imposed by Flexplane will increase from that described above, but will remain an additive factor that is constant across all packets. With larger networks q', the queueing delay in the real network, can also increase, adding more jitter to the end-to-end packet delay, l'.

In multi-rack topologies, real packets must follow the same paths as those taken by their emulated counterparts through the emulated network, in order to avoid causing congestion in the real network. When the implemented resource management scheme does not control packets' paths, this agreement can be achieved by configuring the routing policies in the emulated routers. For example, if ECMP is used in the real network with a known hash function, emulated routers can run ECMP with the same hash function. When the implemented resource management scheme does control packets' paths (as in CONGA [9] or DeTail [54]), packets can be made to follow the prescribed paths using tunneling or ECMP spoofing, as described in [40].

3.5 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.

Emulator APIs. To add a scheme to Flexplane, users can choose between two C++ APIs to implement at the emulator, as shown in Table 1. In the pattern API (the API we expect most users to follow), users implement four functions. These functions divide packet processing into routing (choosing an output port), classification (choosing amongst multiple queues for a given port), enqueuing (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. If the user wants more flexibility than the pattern API, we provide a more general API specified in the generic row of Table 1.

Endpoint APIs. Flexplane also provides a C API for users to read 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 modify packet headers at the endpoints (e.g., to add ECN marks) by implementing 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.

3.6 Scaling the Emulator with Multi-core

Supporting large networks with Flexplane is challenging because it requires high aggregate emulator throughput. The emulator uses multi-core to achieve this. It runs on a dedicated multi-core machine; as long as each additional core increases total throughput, the emulator can scale to large networks by using many cores.

Common approaches to multi-core packet-processing are ill-suited for Flexplane. For example, if Flexplane processed each abstract packet to completion on a single core (similar to RouteBricks [21]), all cores would contend for the shared state of emulated routers and endpoints, limiting scalability. Passing abstract packets down a linear pipeline (as in Fastpass [40]) does not work either, because different packets may access router state in different orders.

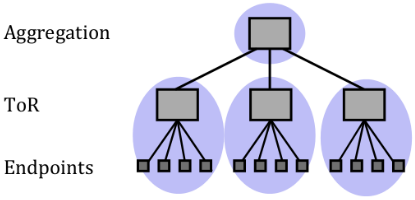


Figure 2: The emulator pins network components (grey) to CPU cores (purple) to avoid sharing state across cores.

Instead, Flexplane's multi-core architecture leverages the fact that networks of routers and endpoints are naturally distributed. Flexplane pins each component of the emulated network topology to a core and passes abstract packets from core to core as they traverse the topology. With this architecture, router state is never shared across cores and cores only communicate when their network components have packets to send to or receive from a different core. This limited communication between cores allows throughput to scale with the number of available cores (§5.3).

To assign emulated components to cores, the emulator distributes the routers amongst the available cores and assigns endpoints to the same core that handles their top-of-rack switch (Figure 2). The Flexplane framework handles inter-core communication with FIFO queues of abstract packets.

For large routers or computationally heavy schemes, one core may not provide sufficient throughput. Future work could explore splitting each router across multiple cores. Different output ports in a router typically have little or no shared state, so one option would be to divide a router up by its output ports and to process different ports on different cores.

Inter-core communication. 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 using CPU cycle counters, 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 so that 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. This is possible because packets are processed in batches;

later packets can be prefetched while the core performs operations on earlier packets. This is similar to the group prefetching technique described in [31].

3.7 Fault Tolerance

Abstract packets. 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.

Emulator. We handle emulator fail-over in the same way as in Fastpass [40]. The emulator maintains only soft state so that a secondary emulator can easily take over on failure. The primary sends periodic watchdog packets to the secondary; when the secondary stops receiving them it takes over and begins emulation. Endpoints update the secondary with their pending abstract packets.

4 Implementation

We implemented Flexplane by extending Fastpass [40]. The Flexplane emulator uses the Intel Data Plane Development Kit (DPDK) [3] for low-latency access to NIC queues from userspace. Support for Flexplane at the endpoints is implemented in a Linux kernel module, which functions as a Linux `qdisc`, intercepting and queueing packets just before they are passed to the driver queue.

We have implemented and experimented with seven different schemes in Flexplane:

DropTail: DropTail queues with static per-queue buffers.

RED: Random Early Detection as described in [24], with and without ECN [23].

DCTCP: ECN marking at the switches, as in [10].

Priority queueing: strict priority queueing across at most 64 queues with classification based on the DSCP field.

Deficit round-robin: round-robin across at most 64 queues, as described in [44], with classification by flow.

HULL: phantom queues as described in [11]. We omit the hardware pacer at the endpoints.

pFabric: pFabric switches as described in [12]. We use standard TCP cubic at the endpoints, omitting the probe mode and other rate control optimizations. We use the remaining flow size as the priority for each packet.

Fastpass [40] relies on clock synchronization using the IEEE1588 Precision Time Protocol (PTP) to ensure that endpoints send packets at precisely the prescribed times. However, we found that it is sufficient to clock the transmissions of real packets using the arrivals of the corresponding abstract packets at the endpoint. This renders PTP unnecessary.

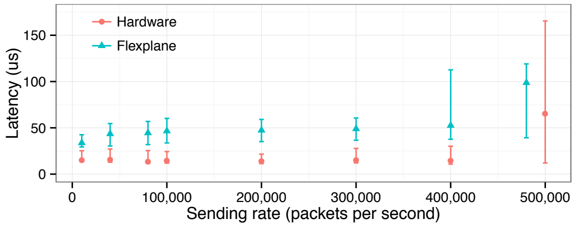


Figure 3: Packet latency under varying loads for one sender in Flexplane and the baseline hardware network. Points show medians; error bars show minimum and 99th percentile observed over 30 seconds.

5 Evaluation

We conduct experiments in a large production network on a single rack of 40 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. 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 in explicit comparisons for emulation accuracy. We use an MTU size of 1500 bytes, and we disable TCP segmentation offload (TSO) (§6.2).

Our experiments address the following questions:

1. **Accuracy**: How well does Flexplane predict the behavior of schemes already supported in hardware?
2. **Utility**: How easy is Flexplane to use and what forms of experimentation can it enable?
3. **Emulator throughput**: Does the Flexplane emulator provide sufficient throughput to support datacenter applications?

5.1 Accuracy

In this section, we evaluate how faithfully Flexplane predicts results obtained with hardware switches, in terms of latency for individual packets, throughput and queue occupancies in the network, and flow completion times observed by applications.

Latency. First we compare the latency of packets in Flexplane to that of packets 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 [5]; the receiver sends back responses. 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 3 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. Note that state-of-the-art software routers add latencies of the same order of magnitude for each hop, even without the added round-trip time to an off-path emulator: 47.6-66.4 µs [21] for a CPU-based software router; 30 µs [31] or 140-260 µs [27] for GPU-based software routers.

Throughput. Next we evaluate accuracy for bulk-transfer TCP, using network-level metrics: throughput and in-network queueing. In each experiment, five machines send TCP traffic at maximum throughput to one receiver.

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. For DCTCP, we use an ECN-marking threshold of 65 MTUs, as recommended by its designers [10].

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.

Queueing. 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 at the same frequency.

Table 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.

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 (§3.4).

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

Flow completion time. Next we evaluate Flexplane's accuracy at the application level in terms of flow completion time (FCT). We run an RPC-based application in which four clients repeatedly request data from 32 servers. The size of the requested data is determined by an empirical workload derived from live traffic in a production datacenter that supports web search (first presented in [10]). It includes a mixture of flows of different sizes. 53% of the flows are small flows of less than 100KB, but 37% of the bytes come from large flows of 10MB or larger. Request times are chosen by a Poisson process such that the clients receive a specified load between 10% and 80%. We normalize the FCT for each flow to the average FCT achieved by a flow of the same size, in an unloaded network, when flows are requested continuously.

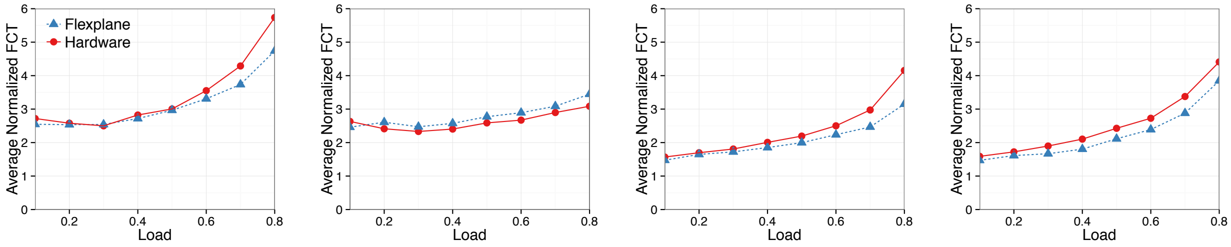
We run this application for DropTail and DCTCP, in Flexplane and in the hardware network. Figure 4 shows the average normalized FCTs. For both small flows and large flows, results with Flexplane closely match results obtained with a hardware network. For loads up to 60% with both schemes, Flexplane estimates average normalized FCTs of hardware to within 2-8% for small flows and 3-14% for large flows. Accuracy decreases slightly for higher loads of 70% and 80%, but remains within 18% for small flows and 24% for large flows.

5.2 Flexplane Utility

In this section, we evaluate the utility of Flexplane. We study how easy it is to write new schemes in Flexplane and provide four examples of how Flexplane can be useful.

Ease of use. To demonstrate the simplicity of implementation, we show the key portions of the source code for priority queueing scheduling in Figure 5. Most schemes require only a few dozen lines of code to implement, as shown in Table 3. pFabric requires significantly more code than other schemes because it does not maintain packets in FIFO order between the enqueue and dequeue stages; 170 of the 251 lines of code are used to implement custom queueing.

Parameter tuning. Flexplane enables users to quickly tune protocol parameters to accommodate different net-



(a) DropTail, (0, 100KB] **(b)** DCTCP, (0, 100KB] **(c)** DropTail, (10MB, ∞) **(d)** DCTCP, (10MB, ∞)

Figure 4: Flexplane closely matches the average normalized flow completion times of hardware for DropTail and DCTCP. The left two graphs show results for small flow sizes; the right two graphs show results for large flow sizes.

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

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

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

Figure 5: Source code for a priority scheduler in Flexplane over ≤ 64 queues. A PacketQueueBank stores packets between the calls to `enqueue` and `schedule`.

scheme	LOC
drop tail queue manager	39
RED queue manager	125
DCTCP queue manager	43
priority queueing scheduler	29
round robin scheduler	40
HULL scheduler	60
pFabric QM, queues, scheduler	251

Table 3: Lines of code (LOC) in the emulator for each resource management scheme.

works. For example, the authors of HULL [11] conducted evaluations using a testbed with 1 Gbits/s links; we use Flexplane to tune HULL's parameters to fit our 10 Gbits/s network. 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, and suggest a marking threshold of 3-5 KB for 10 Gbits/s links. We found, however, that throughput degraded significantly with a 3 KB marking threshold, achieving only 5 Gbits/s total with four concurrent flows. We therefore increased the marking threshold until our achieved throughput was 92% of the drain rate (this is what [11] achieves with their parameters); the resulting threshold is 15 KB. Flexplane helped us conduct this parameter search quickly and effectively.

Evaluating trade-offs. In this example, we demonstrate how one might use Flexplane to evaluate the performance of a specific application with different resource management schemes. We do not argue that any scheme is better than any other, but instead demonstrate that there are trade-offs between different schemes (as described in [47]), and that Flexplane can help users explore these trade-offs.

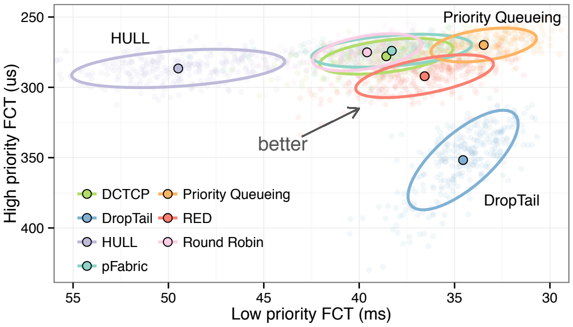
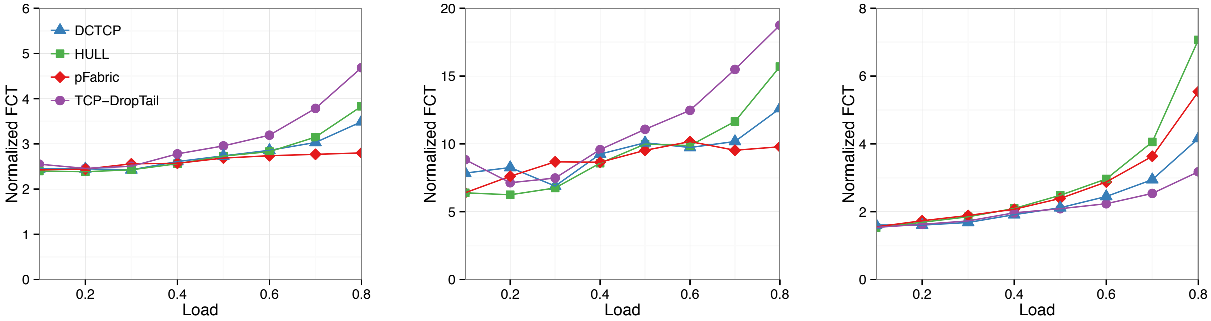


Figure 6: Flexplane enables users to explore trade-offs between different schemes. Large points show averages over the entire experiment, faded points show averages over 1s, and ellipses show one standard deviation. Note the flipped axes.

We use an RPC-based workload and consider the trade-off that schemes make between performance for short flows and performance for long flows. In the experiment, four 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 Mbyte "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 evaluate the schemes discussed in §5.1, as well as TCP-cubic/per-flow-DRR, TCP-cubic/priority-queueing, HULL, and pFabric.

Figure 6 shows the trade-off each scheme makes on this workload. With DropTail, large queues build up in the network, leading to high flow completion times for the high-priority requests. However, DropTail senders rarely cut back their sending rates and therefore achieve good FCTs for the long requests. At the other end of the spectrum, HULL's phantom queues cause senders to decrease their sending rates early, leading to unutilized bandwidth and worse performance for the low priority flows; the high priority flows achieve relatively good performance because they encounter little queueing in the network.



(a) (0, 100KB]: Average **(b)** (0, 100KB]: 99th percentile **(c)** (10MB, ∞): Average

Figure 7: Normalized flow completion times for the web search workload, for four different schemes run in Flexplane. Note the different y axes.

Priority queueing performs well on this simple workload, achieving good performance for both flow types. A network operator could use these results to determine what scheme to run in their network, depending on how they value performance of high priority flows relative to low priority flows.

Real applications. In addition to enabling experimentation with network-bound workloads like the one above, Flexplane enables users to evaluate the performance impact of different resource management schemes on real applications whose performance depends on both network and computational resources. We consider two applications that perform distributed computations using Spark [1]. The first uses block coordinate descent [2] to compute the optimal solution to a least squares problem; this is a staple of many machine learning tasks. The second performs an in-memory sort [4]. For this experiment, we use a small cluster of 9 machines (1 master and 8 workers), each with 8 cores, connected via a single switch with 1 Gbit/s links. We use Flexplane to run each application with DropTail, DCTCP, and HULL.

Table 4 shows that different Spark applications are affected in different ways by a change in resource management scheme. The sort application, which includes multiple waves of small tasks and small data transfers, shows small improvements in completion time, relative to DropTail, when run with DCTCP or HULL. In contrast, coordinate descent takes 4.4% longer to complete when run with DCTCP, and 29.4% longer when run with HULL. This is because this application sends data in a small number of bulk transfers whose throughput is degraded by HULL's, and to a lesser extent DCTCP's, more aggressive responses to congestion. Flexplane enabled us to quickly evaluate the impact of a change in resource management scheme on these real-world applications. Because these applications spend much of their time performing computation (>75%), it is not possible to accurately conduct this experiment in a network simulator today.

	% Change in Completion Time	
	Coordinate descent	Sort
DCTCP	+4.4%	-4.8%
HULL	+29.4%	-2.6%

Table 4: Percent change in completion time of two Spark applications when run with DCTCP or HULL, relative to when run with DropTail.

Reproducible research. Here we demonstrate how experiments that researchers conducted in simulation in the past can be conducted on a real network with Flexplane, and how results in a real network might differ from those in simulation. To do so, we recreate an experiment that has been conducted in several other research papers [12, 14, 26]. We use the same network configuration and workload as in the flow completion time experiment in §5.1; this is the same workload used in prior work.

Figure 7 shows the results of running this workload for DropTail, DCTCP, HULL, and pFabric, in Flexplane, at loads ranging from 10% to 80%. We present the average and 99th percentile normalized flow completion time for small flows, and the average normalized flow completion time for large flows, as in prior work.

We observe the same general trends as in prior work. For the small flows, DropTail performs the worst, with performance degrading significantly at the highest loads and at the 99th percentile. In contrast, pFabric maintains good performance for small flows, even at high load and at the tail. For large flows, DCTCP and DropTail maintain the best performance, while HULL and pFabric degrade significantly at loads of 70%-80%. For HULL, this is because the required bandwidth headroom begins to significantly limit large flows. For pFabric, performance degrades at high load because short queues cause many packets to be dropped. This may be exacerbated by the fact that we do not use all TCP modifications at the endpoints, including the probe mode (which is particularly important at high load).

Our results demonstrate an unexpected phenomenon. One would expect that under low load (e.g., 10%), small flows would achieve a normalized FCT close to

1; previous simulation results have corroborated this intuition [12, 26]. In contrast, our results show that the average normalized FCTs across all schemes begin at around 2.5, even under the lightest load. These results obtained in Flexplane agree with those obtained on the hardware network, for DropTail and DCTCP (Figure 4).

This unexpected behavior is due to the properties of real endpoint network stacks. In real endpoints, application-layer latency depends on the rate at which packets are sent; when packets are sent at a high enough rate, the latency decreases significantly. For example, in our network, the `ping` utility reports average ping latencies of 77 µs when pings are sent every 2 ms; this decreases to 14 µs when pings are sent every 50 µs. Because many of the bytes in this workload belong to large flows, the number of queries per second is relatively small (513 per second to saturate a 10 Gbits/s NIC). The result is that, under most loads, packets are not sent at a high enough rate for small flows to achieve the ultra-low latency achieved when flows are requested continuously; their normalized FCTs are thus much higher than 1. Large flows still approach normalized FCTs of 1 because the FCT is dominated by the transmission time. This behavior would be hard to capture accurately in simulation, but is automatically captured with Flexplane.

5.3 Emulator Throughput

The aggregate throughput of the Flexplane emulator determines the size of network and the types of applications that Flexplane can support. In this section we evaluate how emulator throughput scales with the number of cores and how it varies across resource management schemes.

Workload. For all throughput experiments, we generate a synthetic network load using additional cores on the emulator machine. Sources and destinations are chosen uniformly at random. 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 periodically adjusts the load, increasing it as long as all cores are able to sustain it, and decreasing it when a core falls behind. We report the total throughput observed over the last 30 seconds, aggregated over all routers in the topology. We conduct experiments on a 2.4 GHz Intel Xeon CPU E7-8870 with 10 cores and 32GB of RAM.

Scaling with number of cores. We consider several variants on a simple datacenter network topology. Each topology includes racks of 32 endpoints connected via a ToR switch; ToRs are connected by a single aggregation (Agg) switch. We assign the Agg switch to its own CPU core, and assign each ToR and its adjacent endpoints to another core, as shown in Figure 2. 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 in any given configuration, all routers process the same amount of network throughput. All routers run DropTail.

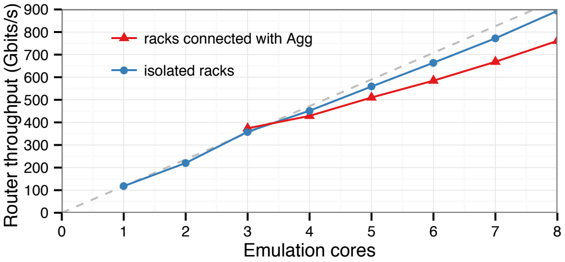


Figure 8: Maximum throughput achieved by the Flexplane emulator for different numbers of emulation cores. The grey dashed line indicates linear scaling based on the throughput achieved with a single emulation core.

The largest topology we measure has seven racks and achieves a total throughput of 761 Gbits/s. Thus, for example, the emulator could support a network of 224 servers sending at an average rate of 2.5 Gbits/s or a network of 56 servers sending continuously at 10 Gbits/s. This is sufficient throughput for small scale datacenter applications with high network utilization, or medium scale applications with lower network utilization.

Figure 8 shows how throughput scales with different numbers of racks in the topology and correspondingly with different numbers of emulation cores. When racks are connected with an Agg switch (red line), throughput falls short of linear scaling (grey line), but each additional core still provides about 77 Gbits/s of additional throughput. To understand the shortfall from linear scaling, we also show the throughput achieved by a simplified topology of isolated racks in which there is no Agg switch and all traffic is intra-rack (blue line). With this topology, throughput scales almost linearly, achieving 112 Gbits/s of added throughput per core on average, 95% of the 118 Gbits/s achieved by a single core. Thus only a small portion (at most 15%) of the throughput shortfall with connected racks is due to unavoidable contention between cores for shared resources such as the L3 cache. The majority of the shortfall is due to communication with the extra Agg core; a more sophisticated mechanism for inter-core communication might help reduce this shortfall.

Throughput across schemes. Figure 9 shows the maximum throughput achieved by each of the schemes we implemented in Flexplane, for a simple topology with one rack of 32 endpoints. DCTCP and DropTail achieve the highest throughput, about 120 Gbits/s. Most other schemes are only slightly more complex and achieve similar throughputs. The most complex scheme, pFabric, achieves 85 Gbits/s, demonstrating that even complex schemes can achieve sufficient throughput in Flexplane.

Comparison with software routers. Flexplane outperforms existing software routers in terms of individual

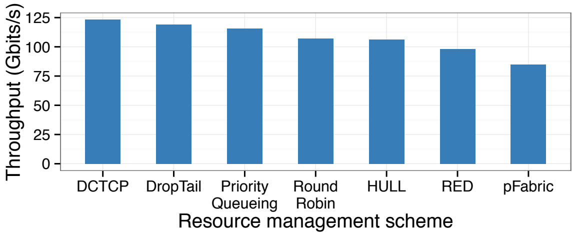


Figure 9: Maximum throughput achieved by the emulator for different resource management schemes with one rack of 32 machines.

router capacity. When emulating a single router with 32 endpoints, Flexplane achieves a total throughput of 118 Gbits/s, compared to a maximum of 35 Gbits/s in RouteBricks [21] and 40 Gbits/s in PacketShader [27], 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 achieves a total router throughput of 761 Gbits/s with 10 2.4 GHz cores (8 emulation cores plus 2 stress test cores). This amounts to 81 times as much throughput per clock cycle in Flexplane.

The difference arises for two reasons. Because Flexplane only processes abstract packets, its memory footprint is relatively small; all of the memory used in the stress test fits in the 30 MB L3 cache shared by the 10 cores. In contrast, RouteBricks applications accessed 400-1000 bytes from memory for each 64-byte packet, likely degrading throughput. In addition, 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.

6 Discussion

6.1 Overheads

Flexplane consumes some network resources; in this section, we quantify these overheads.

Network bandwidth. Communication between endpoints and the emulator consumes some of the network bandwidth. We measured the total throughput loss due to this communication (including abstract packets, acknowledgements, and retransmissions) to be only 1-2% of the total achievable throughput on a 10 Gbits/s link (§5.1).

Emulator NICs. To support larger networks, the emulator may need multiple NICs. Assuming 10 Gbits/s NICs and that traffic to the emulator is about 1-2% of all network traffic, the emulator needs one NIC for every 500-1000 Gbits/s of network throughput provided.

Emulator CPUs. The emulator requires about one communication core per 550 Gbits/s of network throughput and one emulation core per 77 Gbits/s of throughput. This means that every 550 Gbits/s of network throughput requires about 8 cores in the emulator.

6.2 Limitations

Though Flexplane has many uses (§5.2), it has a few limitations. First, Flexplane cannot scale to support arbitrarily large networks, because of throughput limitations at the emulator. Our 10-core emulator supports up to 760 Gbits/s of throughput (§5.3), sufficient for experiments spanning a few racks with 10 Gbits/s links, but insufficient for large-scale experiments involving dozens of racks. Second, in order to provide high performance, Flexplane maintains a fixed abstract packet size (§3.1); this may degrade accuracy with schemes whose behavior depends on packet size (e.g., fair queueing [20]) under workloads with diverse packet sizes. Third, because Flexplane adds some latency overhead (§3.4), it is not suitable for experimentation with schemes that display drastically different behavior with small changes in network latency. Finally, in order to faithfully emulate in-network behavior, Flexplane requires the ability to control the transmission time of each individual packet. This means that TCP segmentation offload (TSO) must be disabled in order to use Flexplane. Without TSO, many network stacks are unable to saturate high speed links (e.g., 10 Gbits/s and faster) with a single TCP connection; a multi-core stack may overcome this limitation.

7 Related Work

Existing approaches to programming and experimenting with resource management schemes fall into three broad categories: simulation (as discussed in §2.1), programmability in software, and programmable hardware.

7.1 Programmability in Software

Software routers. Software routers such as Click [18, 33], Routebricks [21], PacketShader [27] and GSwitch [52] process packets using general-purpose processors or GPUs, providing similar flexibility to Flexplane. However, Flexplane requires much less CPU and network bandwidth – a fiftieth to a hundredth of each – to achieve the same router throughput. Other software approaches [8, 35, 43] supplement hardware with programmable elements, but face the same throughput limitations because they must process packets "on-path" with CPUs for full flexibility.

End host approaches. Eden provides a programmable data plane for functions that can be implemented purely at the end hosts [15]. Unlike Flexplane, it cannot support schemes that require in-network functions beyond priority queueing, such as D^3 [53], PDQ [29], or pFabric [12].

7.2 Programmable Hardware

Several approaches such as NetFPGA [34], RiceNIC [42], CAFE [36], Chimpp [41], Switchblade [13], and [47] use FPGAs to enable programmable header manipulations or resource management. Though these approaches provide higher performance than software approaches, programming an FPGA is typically much harder than programming Flexplane in C++. In addition, a network must be equipped with FPGAs to benefit from such approaches.

Other work targets programmable switching chips. The P4 language [17] aims to provide a standard header manipulation language to be used by SDN control protocols like OpenFlow [38]. However, P4 does not yet provide the primitives necessary to support many resource management schemes, and, as a domain-specific language, P4 falls short of the usability of C++. The Domino language [45] allows users to express resource management schemes in a C-like language, which the Domino compiler then compiles to programmable switches. Domino approaches the flexibility of Flexplane, but users must upgrade to new programmable switches in order to reap the benefits. In contrast, Flexplane provides a way to experiment in existing network infrastructure.

8 Conclusion

In this paper we presented Flexplane, an experimentation platform for users to program resource management algorithms in datacenter networks. We demonstrated that Flexplane accurately reproduces the behavior of schemes already supported in hardware, sustains aggregate throughput of up to 760 Gbits/s with a 10-core implementation, and enables experimentation with schemes not yet supported in commodity routers. Flexplane offers a practical alternative to simulation for researchers, a way of evaluating new protocols for network operators, and a platform for experimenting in real networks for students.

Acknowledgements

We thank Omar Baldonado and Sanjeev Kumar of Facebook for their enthusiastic support of this collaboration, Vadim Balakhovski of Mellanox for his generous assistance with Mellanox drivers and equipment, Devavrat Shah for useful discussions, Kay Ousterhout and Shivaram Venkataraman for their assistance in running Spark applications, and our shepherd Anja Feldmann and the NSDI reviewers for their useful feedback. Ousterhout was supported by an NSF Fellowship and a Hertz Foundation Fellowship. Perry was supported by a Facebook Fellowship. This work was funded in part by NSF Grant CNS-1526791. We thank the industrial members of the MIT Center for Wireless Networks and Mobile Computing for their support and encouragement.

References

[1] Apache Spark. http://spark.apache.org/.

[2] Distributed Matrix Library. https://github.com/amplab/ml-matrix.

[3] DPDK Boosts Packet Processing, Performance, and Throughput. http://www.intel.com/go/dpdk.

[4] MemorySortJob. https://github.com/kayousterhout/spark-1/blob/monoDeadline/examples/src/main/scala/org/apache/spark/examples/monotasks/MemorySortJob.scala.

[5] Network Benchmarking Utility. https://github.com/mellanox/sockperf.

[6] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/index.html.

[7] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, and H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3):6–18, 2002.

[8] M. Al-Fares, R. Kapoor, G. Porter, S. Das, H. Weatherspoon, B. Prabhakar, and A. Vahdat. NetBump: User-extensible Active Queue Management with Bumps on the Wire. In *ANCS*, 2012.

[9] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[11] 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.

[12] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*, 2013.

[13] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. In *SIGCOMM*, 2010.

[14] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*, 2015.

[15] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea. Enabling End-host Network Functions. In *SIGCOMM*, 2015.

[16] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. In *SIGCOMM*, 1996.

[17] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR*, July 2014.

[18] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *ATC*, 2001.

[19] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *SIGCOMM*, 1998.

[20] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM CCR*, Sep. 1989.

[21] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *SOSP*, 2009.

[22] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *SIGCOMM CCR*, Jan. 2006.

[23] S. Floyd. TCP and Explicit Congestion Notification. *SIGCOMM CCR*, Oct. 1994.

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

[25] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[26] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *NSDI*, 2015.

[27] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-Accelerated Software Router. In *SIGCOMM*, 2010.

[28] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena. Network Simulations with the ns-3 Simulator. *SIGCOMM demonstration*, 2008.

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

[30] L. Jose, L. Yan, M. Alizadeh, G. Varghese, N. McKeown, and S. Katti. High Speed Networks Need Proactive Congestion Control. In *HotNets*, 2015.

[31] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *NSDI*, 2015.

[32] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[33] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *TOCS*, 18(3):263–297, 2000.

[34] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education*, 2007.

[35] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *NSDI*, 2011.

[36] G. Lu, Y. Shi, C. Guo, and Y. Zhang. CAFE: A Configurable Packet Forwarding Engine for Data Center Networks. In *PRESTO*, 2009.

[37] P. E. McKenney. Stochastic Fairness Queuing. In *INFOCOM*, 1990.

[38] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, Apr. 2008.

[39] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal Packet Scheduling. In *NSDI*, 2016.

[40] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*, 2014.

[41] E. Rubow, R. McGeer, J. Mogul, and A. Vahdat. Chimpp: A Click-based Programming and Simulation Environment for Reconfigurable Networking Hardware. In *ANCS*, 2010.

[42] J. Shafer and S. Rixner. RiceNIC: A Reconfigurable Network Interface for Experimental Research and Education. In *ExpCS*, 2007.

[43] A. Shieh, S. Kandula, and E. G. Sirer. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *HotNets*, 2010.

[44] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3), June 1996.

[45] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[46] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*, 2016.

[47] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *HotNets*, 2013.

[48] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *SOSP*, 2001.

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

[50] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-Aware Datacenter TCP (D^2TCP). *SIGCOMM*, 2012.

[51] A. Varga et al. The OMNeT++ Discrete Event Simulation System. In *ESM*, 2001.

[52] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multi-Layer Packet Classification with Graphics Processing Units. In *CoNEXT*, 2014.

[53] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[54] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. *SIGCOMM*, 2012.