Network architecture in the age of programmability

Anirudh Sivaraman[*], Thomas Mason[*], Aurojit Panda[*], Ravi Netravali[+], Sai Anirudh Kondaveeti[*]
[*]New York University, [+]University of California at Los Angeles
anirudh@cs.nyu.edu, tem373@nyu.edu, apanda@cs.nyu.edu, ravi@cs.ucla.edu, sak797@nyu.edu

ABSTRACT

Motivated by the rapid emergence of programmable switches, programmable network interface cards, and software packet processing, this paper asks: given a network task (e.g., virtualization or measurement) in a programmable network, should we implement it at the network's end hosts (the edge) or its switches (the core)? To answer this question, we analyze a range of common network tasks spanning virtualization, deep packet inspection, measurement, application acceleration, and resource management. We conclude that, while the edge is better or even required for certain network tasks (e.g., virtualization, deep packet inspection, and access control), implementing other tasks (e.g., measurement, congestion control, and scheduling) in the network's core has significant benefits—especially as we raise the bar for the performance we demand from our networks.

More generally, we extract two primary properties that govern where a network task should be implemented: (1) *time scales*, or how quickly a task needs to respond to changes, and (2) *data locality*, or the placement of tasks close to the data that they must access. For instance, we find that the core is preferable when implementing tasks that need to run at short time scales, e.g., detecting fleeting and infrequent microbursts or rapid convergence to fair shares in congestion control. Similarly, we observe that tasks should be placed close to the state that they need to access, e.g., at the edge for deep packet inspection that requires private keys, and in the core for congestion control and measurement that needs to access queue depth information at switches.

CCS CONCEPTS

• **Networks → Programmable networks**;

KEYWORDS

Network architecture, programmable networks

1 INTRODUCTION

To keep pace with an ever-changing set of requirements, network infrastructure has rapidly become programmable across the board: from switches to network interface cards (NICs) to middleboxes. For instance, many switch ASIC manufacturers have developed and commercialized high-speed programmable switch ASICs [1–3, 5, 6, 8, 14]. There has also been a recent move towards programmable NICs [7, 9, 13, 30]. Lastly, for some time now, there has been a rise in software packet processing for implementing network functions such as deep packet inspection and firewalls [31, 51, 56].

Programmable networks can potentially benefit chip manufacturers, equipment vendors, and network operators. For a chip manufacturer such as Broadcom or Mellanox, programmable switch and NIC ASICs simplify ASIC design and save hardware design costs. They do so by promoting design reuse and amortizing hardware design cost: a small set of repeatable and programmable hardware primitives can replace a much larger number of individually designed features. Additionally, the same programmable ASIC can be customized to different deployment scenarios. For an equipment vendor such as Arista or Dell, programmability can be used to both rapidly fix bugs in shipping switches and to add new features in firmware without having to ask the chip manufacturer for new hardware. These engineering benefits alone (design reuse, rapid bug fixes, and the ability to reuse a single programmable ASIC) make programmable networks compelling.

At the same time, while it is clear that programmability is useful to both chip manufacturers and equipment vendors, it is still unclear how programmable networks will affect network operators, e.g., Google, AT&T, or Microsoft. Recent academic and industrial work has demonstrated several possibilities enabled by programmable NICs [18, 26, 41, 43, 52] and programmable switches [19, 35, 36, 50, 54]. However, we still do not have the answer to a basic question: *if the entire network is programmable and a network operator is given a network task such as network virtualization or measurement, should the task run at the end hosts (the edge) or the switches (the core)?*

In this paper, we address this question qualitatively by comparing two architectures for implementing network and application tasks: (1) when both the edge (end hosts, NICs, virtual switches, and middleboxes) and the core (switches, and routers) of a network are programmable and (2) when only the edge is programmable. We analyze the effectiveness of each architecture by considering several example tasks: network measurement, resource management (flow scheduling, congestion control, load balancing, and active queue management (AQM)), network virtualization, network security, deep packet inspection, and application acceleration (§2). If a network task can be implemented at the edge without substantial drawbacks (e.g., inaccurate measurements, large processing overheads), we prefer the edge. This is due to the edge's relative ease of programming and abundance of computation and storage resources.

Our key findings are:

(1) Tasks that access state resident on the edge must run on the edge out of necessity. One example is deep packet inspection

(DPI) for spam filtering or worm detection. Here, if the payload is encrypted, the key to decrypt it is typically available only at the edge.

(2) Similarly, tasks that access state stored in the core are more efficiently implemented in the core. For example, measuring the loss rate or queuing delay of a particular switch is more effective when done directly on that switch, as compared to inaccurate end-host approaches like network tomography that incur overheads from pooling together many indirect end-to-end measurements from across the network. Similarly, congestion-control algorithms like DCTCP [16] and XCP [39] illustrate the benefit of direct and accurate congestion feedback from switches (which can directly access queue size information) relative to inferring congestion from end-to-end measurements of packet losses or round-trip times.

(3) Tasks that require large amounts of memory or computing per packet (e.g., network virtualization [27, 29, 42]) can be more easily implemented on the edge than the core. This is because, relative to the core, the edge has more memory resources (DRAM vs. SRAM) and its packet processing requirements are less demanding.

(4) Tasks that must run at very short timescales (e.g., measurement of datacenter microbursts, rapid convergence to a flow's fair rate allocation, and rapid load balancing in response to link/switch failures) need a programmable core. By contrast, tasks that run over longer timescales (e.g., asymptotic TCP fairness guarantees, eventual detection of switch hardware bugs, and long-term averages of queuing latency) can be implemented using the edge alone. We believe that understanding and improving network performance at short timescales will be increasingly important as we move beyond improving average-case performance of a network to improving performance at high percentiles.

(5) Offloading application tasks into the network [35, 36] has benefits—but only in specific settings. While a few instances of niche applications at a small scale can benefit from such offloading, limited switch memory soon prevents multiple applications from offloading tasks into the switch simultaneously.

Reflecting on the findings above, we observe that the placement of a network task is influenced by two factors: (1) the state it needs (e.g., DPI at the edge and measurement at the core) and (2) the timescale at which it runs (e.g., switch bug detection at the edge and microburst detection at the core). These two guidelines can help network operators decide the right location for a network task in a network with programmable switches. We hope our initial findings prompt further research into defining a more comprehensive set of guidelines and a precise taxonomy for determining where network tasks should ideally be implemented in a programmable network. Source code for the simulations used to generate this paper's figures is available at https://github.com/tem373/Network_Tomography_Project.

2 NETWORK TASKS: EDGE VS. CORE

2.1 Terminology and problem setup

For the purpose of this paper, the *edge* of the network refers to both the first-hop network devices that applications use to connect to the network (e.g., access routers, network interface cards, virtual switches, and cellular base stations) and the devices at the boundary between two autonomous systems (e.g., middleboxes and border routers). Everything else (e.g., leaf, spine, and aggregation switches and core routers), we consider part of the *core*. Our definition is borrowed from CSFQ [60] and Fabric [24]. By this definition, a packet can traverse multiple edges and cores on the path from its source to its destination. Because of the widespread prevalence of virtual switches that run on end-host hypervisors, we include the end hosts themselves (e.g., laptops, servers, mobile phones, etc.) as part of the edge.

An equivalent way to distinguish the edge and the core is based on differences in implementation owing to the difference in forwarding rates. For any autonomous system, the number of edge devices is typically much more than the number of core devices; so, the throughput required for each edge device is relatively modest compared to a core device. Hence, an edge device is predominantly implemented using general-purpose software for flexibility—with a small amount of additional hardware for improved speed, either through programmable NICs [7, 9] or bump-in-the-wire FPGAs [30]. A core device is predominantly implemented using special-purpose ASICs optimized for forwarding speed—with a small amount of programmability to accommodate new requirements [1–3].

We also assume that the edge and the core are under the control of a single administrative entity, e.g., (1) a datacenter network operator that controls both the servers (edge) and switches (core) within the datacenter or (2) an Internet Service Provider that controls both the border routers and middleboxes (edge) and core routers (core) within a particular autonomous system. This paper only considers the technical problem of choosing between the edge and core within a single administrative entity. Although equally important, it does not consider the economic problem of compatibility between different administrative entities.

We compare two network architectures: (1) an edge+core architecture in which all edges and cores can be programmed and (2) an edge-only architecture in which only the edges of the network are programmable, with the cores being fixed and responsible only for best-effort packet forwarding. For the edge+core architecture, we assume core programmability similar to recent hardware designs for programmable switches [21, 58, 59]. This programmability permits limited transformations of packet headers (but not payloads) [21], manipulation of a limited amount of state during packet processing [58], and programmable scheduling on a limited number of active flows [59].

This is optimistic because programmable switches vary in their capabilities and not all current programmable switches support all of these capabilities. We adopted this approach because the hardware architectures and instruction sets of programmable switches are in flux and will evolve to meet the needs of network operators. For the edge, we assume relatively abundant availability of main memory in DRAM and the ability for software packet processing (possibly augmented with some hardware such as programmable NICs or FPGAs) to keep up with edge-device line rates.

We carry out our analysis by picking a set of commonly seen network tasks, drawn from network measurement, resource management, network virtualization, deep packet inspection, and application acceleration. We then compare the two architectures above in terms of whether they can implement each task effectively. If programming the edge allows us to implement a task satisfactorily, we prefer this because of the ease of programming the edge—even if an implementation in the core might be slightly more efficient. In other words, for each task, our goal is to understand whether the task really requires a programmable core (and hence the edge+core architecture) in order to be implemented effectively or whether an edge-only architecture suffices.

For simplicity, we focus on implementing a task exclusively at either the edge or the core, without allowing for the possibility of splitting the same task across the edge and the core. While this simplifies our analysis, it does not meaningfully change our conclusions. Some tasks may be complex enough to warrant splitting the task across both the edge and the core. In such cases, we could break up these tasks into sub tasks and then decide whether each sub task should run exclusively on the edge or the core. Our analysis is still applicable to the constituent sub tasks.

2.2 Measurement

We start with the problem of network measurement using an edge-only architecture vs. an edge+core architecture. We proceed gradually, starting with measurement tasks that can be readily implemented using an edge-only architecture. We then discuss measurement tasks where much can be gained from implementing them in the core.

As an example of a measurement task that can be implemented using an edge-only architecture, consider the problem of detecting switch-silent packet drops tackled by Pingmesh [33] and Everflow [64]. These are switch packet drops that leave no trace on the switch, i.e., the loss counter on the switch doesn't log them. They typically arise due to hardware or firmware bugs, e.g., a corrupted TCAM entry in an IP lookup table might cause a black hole. These drops are long-lived, lasting until the faulty switch is rebooted or replaced. Setting up a mesh of infrequent server-to-server probes (e.g., Pingmesh issues at most one probe every 10 seconds between any server pair) suffices to localize the faulty switch, assuming that detection times on the order of minutes are acceptable.

While edge-only solutions like Pingmesh work for timescales on the order of a few minutes, they are insufficient for shorter timescales. The shorter the timescale of measurement, the stronger the case for implementing measurement within the network's core. For instance, consider the problem of measuring the packet loss rate or the queuing delay distribution of a switch deep inside the network. For distributions and loss rates that do not change over an extended period of time, this problem can be solved using the edge-only approach of network tomography, which analyzes end-to-end measurements of losses [22] or delays [53] to indirectly infer link-level loss rates or delay distributions.

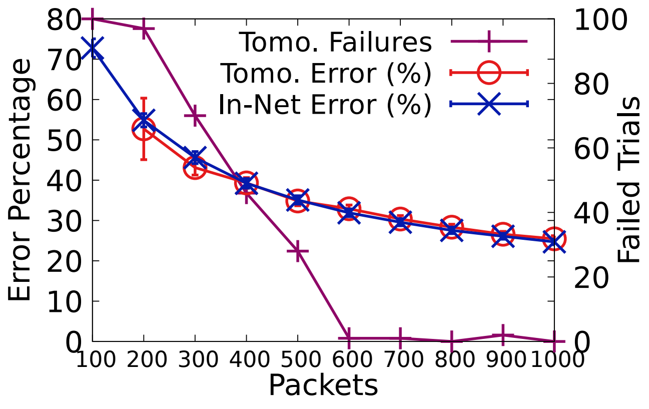
However, tomography has three drawbacks relative to direct in-network measurement at programmable switches. First, it requires additional multicast or unicast probes to continuously transit the queues being measured; if switches treat these probes differently from data packets, the results from tomography could be misleading. Second, the link capacity consumed by these probes is directly proportional to the measurement frequency. Third, the number of samples required to estimate quantities to a certain accuracy is higher using tomography relative to in-network measurement because of the indirect nature of inference in tomography. This implies that tomography cannot effectively estimate rapidly changing loss rates or delay distributions because the loss rate or delay distribution may have changed before sufficient number of samples have been collected and analyzed.

We illustrate the third drawback using a simulation on a binary tree network topology of depth 5. We compare loss tomography [22] against in-network loss estimation (i.e., dividing the number of lost packets by the total number of packets on each link) for a range of sample sizes measured in packets. We assume Bernoulli losses with probabilities 1% and 30%. While 30% is high when averaged over a long time period, losses in many datacenters tend to occur in microbursts [63]. A microburst is a very short period of time (10–100 microseconds) with a surge in link utilization and hence queue size and packet drops. Microbursts are characterized by large loss rates over short intervals separated by large periods with very low loss rates. Thus, such temporarily high loss rates are conceivable as we reduce the timescale of measurements and attempt to measure the loss rate within a microburst.

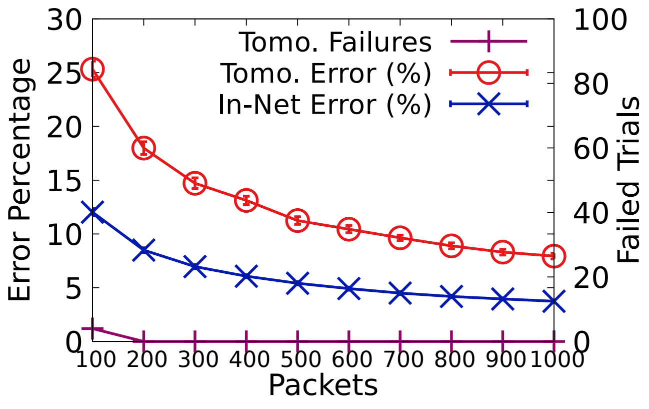
Figure 1 shows the error rates of both tomography and in-network loss estimation when the number of samples is varied. Each data point is an average of 100 trials. The error rates of both approaches increases when the number of samples is decreased, but the in-network approach has lower errors, especially when the loss rate is high (30%). When the loss rate is low (1%), the errors of both schemes are comparable. However, as the number of samples reduces, tomography has to discard several trials because there are not enough packet drops in these trials to meaningfully run inference; this manifests itself as a larger confidence interval. The benefits of in-network estimation relative to tomography are more pronounced when the losses are drawn from a bursty loss process [4] instead of a Bernoulli process. In summary, tomography is well-suited to measuring non-bursty and low loss rates over a long time interval, but not bursts of high loss rates over a short time interval, i.e., microbursts.

Measurements at short timescales are increasingly important as our goal shifts from average performance (e.g., throughput) to tail performance (e.g., tail flow completion time [62]). For instance, microbursts temporarily increase queuing delay and cause packet drops, both of which hurt tail flow completion time. Similarly, measurements of queue depths at short timescales help us determine if load balancing at short timescales is effective. For instance, if the queue on one leaf-to-spine link is much longer than that on another leaf-to-spine link even for a small duration of time, it means that load balancing isn't very effective, which results in increased tail flow completion time.

We note that programming measurement tasks on a switch is limited by on-chip memory. Hence, storing fine-grained measurements (e.g., at the level of 5-tuples) can be challenging. Approaches like Marple [49] that split the measurement task between scarce



(a) 1% loss rate (Bernoulli process)



(b) 30% loss rate (Bernoulli process)

Figure 1: In-network loss measurement has lower errors than tomography.

on-chip switch SRAM and abundant off-chip DRAM address this problem; however, only specific classes of measurement operations can be implemented in this split fashion. In general, the operator should be aware of whether the switch memory can accommodate the working set of flows being measured. In environments like datacenters, this working set is small and can fit into on-chip memory [20]. In others, such as a core router [12], this may not be possible and approaches like Marple might be necessary.

2.3 Resource management

We now consider the problem of resource management: allocating the capacity of a network's resources (e.g., link capacity or buffer space) to competing entities. We use the term resource management to unify different network tasks that ultimately have the same end goal of dividing up a scarce resource: congestion control, packet scheduling, data-plane load balancing, and buffer management algorithms.

Resource management can benefit considerably from an edge+core architecture. For instance, DCTCP [16] provides substantially reduced queuing latency relative to TCP New Reno (an edge-only approach) by using explicit congestion notification (ECN) support in switches to signal congestion to end hosts much before packets start getting dropped. pFabric [17] makes use of fine-grained priority scheduling in switches to provide lower flow completion times relative to DCTCP. NUMFabric [48] and PERC [37] provide much faster convergence of flows to their fair rate allocations by leveraging programmable switches. Several other resource management algorithms that need switch support are also enabled by an edge+core architecture (e.g., DeTail [62], XCP [39], and RCP [61]).

Similar to measurement inside the core of the network, resource management algorithms that exploit core programmability will become increasingly relevant as we demand more from our networks, by pushing networks to have higher utilization and by optimizing for tail statistics (e.g., tail flow completion time) instead of average-case performance.

Resource management algorithms with in-network components perform better relative to their edge-only counterparts because they are able to take advantage of explicit feedback on the extent of congestion in the network rather than indirectly inferring congestion from end-to-end measurements. For instance, XCP and RCP both use multi-bit signals to signal congestion from the core to the end hosts. Similarly, DCTCP uses a single-bit ECN mark. By contrast, algorithms in an edge-only architecture have to implicitly infer congestion at a particular switch by using the loss of a packet [34], changes in end-to-end round-trip time [47], or changes in received rate [23].

Implementing resource management algorithms on switches also faces some challenges. Algorithms that do not maintain per-flow state (e.g., XCP) are considerably easier to implement because of their less demanding memory requirements. However, some of these algorithms require floating point operations (e.g., XCP), which are not yet supported in their most general form on programmable switches. Algorithms that do require per-flow state on switches (e.g., WFQ [28]) are constrained by the limited number of active flows supported by programmable schedulers [59]. Approaches that approximate floating point operations on switches [54] or approximate programmable scheduling using a small number of fixed queues [55] hold considerable promise until the instruction sets of these switches become more powerful.

2.4 Deep packet inspection

Deep packet inspection like spam filtering and worm detection needs to access the payloads of packets. Such tasks need to run at the edge for two reasons. First, the payload may be encrypted and only an end host might possess the key to decrypt it. Second, extracting and processing the payload is a much more demanding operation than packet header processing, especially at an aggregate forwarding rate of a few Tbit/s. Hence, programmable switches typically don't support payload inspection or modification [21].[1]

[1]Approaches based on homomorphic encryption suffer from low throughput (less than 200 Mbit/s currently [57]). This throughput suffices for some middleboxes at the edge (e.g., intrusion detection), but it is far too slow for the core.

2.5 Network security

We now consider defending against attacks on network resources of an end host's networking stack. One example is the Slowloris attack. Here, an attacker opens several TCP connections to a victim and transfers little to no data on each connection. In the process, the attacker exhausts the victim's limit on the number of concurrent connections. The TCP SYN flood attack similarly exhausts the victim's limit on the number of half-open TCP connections. In the examples above, the definition of an attack is application-dependent: many web sites legitimately require a browser to open several concurrent connections that each transfer a small amount of data. As a consequence, the defenses are also typically deployed at the end hosts themselves (e.g., SYN cookies [11]). In summary, when the resource being attacked is a resource within the end-host's networking stack, the defense is best left to the end-host/edge.

2.6 Network virtualization

Network virtualization [27, 29, 42] is the ability to provide independent tenants in a multi-tenant datacenter the illusion that their VMs are running on their own dedicated network, similar to an on-premise network. Network virtualization, as NVP [42], VFP [29], and Andromeda [27] show, can be implemented using a virtual switch such as Open vSwitch running on the hypervisor of each server. This virtual switch enforces per-tenant policies using a logical packet-forwarding pipeline and carries out all required packet transformations in the virtual switch. Finally, the packet is forwarded to its destination hypervisor using tunnels that have been setup between every pair of hypervisors. The core of the network is thus restricted to IP forwarding, while all of the network virtualization logic itself resides on the edge's virtual switches. As Andromeda [27] observes, this approach is preferable to implementing virtualization in the core because the availability of relatively abundant CPU memory on the edge allows it to easily scale to a large number of virtual networks.

2.7 Application acceleration

A few recent projects [35, 36] offload application tasks onto Barefoot Networks' programmable Tofino switching chip. These projects demonstrate that significant performance benefits can be realized by moving application tasks onto a programmable switching chip. For concreteness, we discuss two projects that have mature implementations with conclusive performance benefits: NetCache [36] (a load-balancing cache implemented on a single switch) and NetChain [35] (a fault-tolerant lock service implemented on a set of switches). Despite their clear performance benefits, NetCache and NetChain both have a few shortcomings, which makes the role of a programmable core in general-purpose application acceleration unclear. We discuss these shortcomings below.

First, both NetCache and NetChain are restricted to niche functionality. NetCache [36] provides a cache with around a 50% hit rate. This hit rate is sufficient for balancing load on backend servers, but not as a general-purpose cache, which typically supports hit rates around 90%. The lower hit rate is a result of limitations on the amount of switch memory. NetChain [35] implements a fault-tolerant key-value store on switches. The number of entries in this key-value store is limited by on-chip switch memory. The size of the values in the key-value store is limited to a few hundred bytes due to limitations on how much data the switch can process every clock cycle. As a result, NetChain is good for applications that store a small number of keys with a small value size (e.g., a lock service), but its general applicability is unclear.

Second, in a setting where different applications share the programmable memory of a switch, this memory will become a scarce resource. This problem becomes even worse under two conditions: (1) increasing single-switch port counts that allow more servers (and thus applications) to reside under the same switch and (2) as we go up the layers from leaf to spine to aggregation in a datacenter network. Unlike network tasks (e.g., resource management) that are broadly useful to several applications, application tasks are by definition only useful to specific applications. Consuming switch memory for application tasks, at the cost of reduced memory for network tasks is the wrong architectural choice. There are settings that might permit this (e.g., when the routing tables are small), but again the general applicability is unclear.

Third, at this point, NetCache [36] is limited in scale to a single rack. NetChain [35] is limited to a few racks. In both cases, it is currently unclear how the systems can scale to larger sizes spanning an entire datacenter given the limited processing and storage resources on a switch.

As an aside, NetCache and NetChain show that a switching ASIC can be a hardware accelerator for niche domains (high communication and low computation needs [35]) beyond networking—reminiscent of the early days of GPGPU computing. However, if this is the intent, to maximize performance, the switching ASIC should ideally be integrated with the server CPU on the same motherboard. If that happens, we would then classify the switching ASIC as a hardware accelerator that is part of the edge, much the same way as an FPGA, GPU, or Tensor Processing Unit [38] that is closely integrated with a server machine.

3 ALTERNATIVE ARCHITECTURES

The edge-only and edge+core architectures are two points on a spectrum of architectures. Here, we briefly consider some alternatives to the edge-only and edge+core architectures, showing how they don't change our basic conclusions from the previous section.

3.1 Universal architectures

There are hybrid architectures that combine edge-only programmability with a smarter, but fixed, core. These architectures augment the best-effort packet forwarding capability of the network's core with a small set of fixed, *universal* data-plane features to provide flexibility using just edge or control-plane programmability. Two examples are (1) universal packet scheduling (UPS) [46], a universal scheduler to emulate other schedulers, and (2) UnivMon [45], a universal sketch for several measurement questions.

This hybrid approach would be highly desirable *if* there was a small set of universal features that provided sufficient expressive power using edge programmability alone, justifying the hardening of these universal features in switch silicon. This would provide a preferable alternative to designing programmable switches. However, current results on such hybrid architectures (UPS and UnivMon) suggest that they do not yet provide sufficient expressive power. Furthermore, they only apply to specific domains such as packet scheduling and measurement.

Even within the space of scheduling and measurement for which UPS and UnivMon are designed respectively, UPS and UnivMon fall short. The universality of UPS has only been demonstrated in theory under two very strong conditions: (1) the UPS scheduler needs upfront access to the entire trace of scheduled packets of the scheduler it is trying to emulate and (2) there can be at most two bottleneck queues in the network. In practice, the space of scheduling algorithms that UPS can emulate is further limited relative to what the theory predicts. For instance, UPS cannot emulate traffic shaping and hierarchical schedulers. These limitations are a direct consequence of UPS assuming a non-programmable core and do not reflect a limitation of the UPS algorithm itself. In the case of UnivMon, the use of sketches limits UnivMon to count/volume-based statistics alone. For instance, UnivMon cannot be used to maintain an exponentially weighted moving average or count the number of packets in a TCP flow that were received out-of-order.

It is possible that a universal switch mechanism will be discovered, obviating the need for switch programmability. For now, however, such a "silver bullet" feature set covering a broad variety of network tasks remains elusive. Further, the list of demands on switches is on the upswing (e.g., new tunneling formats [10] or new measurement support [40]). When demands—and hence switch features—are in flux, programmability tackles change head on and provides a higher degree of future proofness than universal mechanisms.

3.2 Other architectures

DumbNet [44] is a network architecture that employs simple stateless switches and moves management complexity to end hosts. End hosts use source routing to route packets by providing a list of tags in each packet that directly identify output ports at each switch without the need for any lookups. A minimal switch simplifies management, but it gives up significant performance and monitoring benefits, as we have shown earlier (§2).

Some network architectures [25, 32] advocate the use of programmable SmartNICs to provide a programmable network.[2] We view the rise of SmartNICs [7, 9, 30] as complementary and beneficial to our view of leveraging programmability in both the edge and the core. Within our terminology, SmartNICs are part of the edge and provide a way to scale up edge packet processing as we move to NIC speeds of 100 and 400 Gbit/s, while still providing flexibility. However, SmartNICs alone are insufficient for all network tasks: as §2 shows, there are benefits to having a programmable core in addition to a programmable edge.

4 GUIDELINES FOR PLACING TASKS

We now take a step back from our specific examples in §2 and attempt to extract guidelines that govern where (edge vs. core) a network task should be placed. The first guideline is that of timescales. When the timescale of the network task is long (e.g., eventual detection of a switch hardware bug [33], long-term fairness across flows [34], and long-term measurements of steady-state delay distributions or loss rates [22, 53]), an edge-only architecture suffices. When the timescales are short (e.g., fine-grained load balancing [15], rapid convergence of a flow to its fair rate [37], and detection of fleeting microbursts [63]), an edge+core architecture has significant benefits.

The second guideline is that of locality: functionality should be implemented closest to the state that it needs. We see this in the case of deep packet inspection tasks running at the edge and in the case of congestion control and measurement tasks running at the core. In each example, the task is implemented at the location where the state required for the task is actually available (i.e., decryption keys at the edge and queue depth information in the core).

5 CONCLUSION AND OUTLOOK

At the beginning, we sought to answer the question: if the network is fully programmable, given a network task, where should it be implemented? By analyzing several common network tasks, we arrived at two guidelines that govern where a network task should be implemented: the time scale at which the task runs and what data the task needs to read and/or write.

We hope this paper is just the beginning of a broader conversation around developing a taxonomy of network tasks based on where in a programmable network (edge vs. core) they should be implemented. Ideally, we would have a theory of network task placement that, given an unambiguous specification of a network task, would accurately tell us where the network task should be implemented. We are far from such a theory, but our qualitative analysis suggests that there might be general task properties (like time scales and locality)—which transcend specific example tasks—that govern where in the network a task should be implemented.

Developing such a theory is an interesting area of future work that has both intellectual and practical value. Intellectually, such a theory might allow us to precisely understand both the inherent nature of network tasks and innate differences between different tasks. Practically, such a theory might allow a compiler to automatically translate from a specification of the network task to an implementation at the appropriate network location predicted by the theory. We hope this paper and its results initiate a more quantitative and formal investigation into a theory of network architecture (i.e., what task should run where) for programmable networks.

REFERENCES

[1] Barefoot: The World's Fastest and Most Programmable Networks. https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/.
[2] Broadcom Ships Jericho2: Driving the Merchant Silicon Revolution in Carrier Networks. https://people.ucsc.edu/~warner/Bufs/CSG-DNX-Switching-J2%20Feb%2016%202018.pdf.
[3] Broadcom Trident 3 - Programmable, Varied And Volume. http://packetpushers.net/broadcom-trident3-programmable-varied-volume/.
[4] Burst error - Wikipedia. https://en.wikipedia.org/wiki/Burst_error.
[5] Cisco live! June 25–29 2017, Las Vegas, NV. http://clnv.s3.amazonaws.com/2017/usa/pdf/BRKARC-3467.pdf.
[6] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.
[7] Mellanox BlueField SmartNIC 25Gb/s Dual Port Ethernet Network Adapter. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.
[8] Mellanox Spectrum-2 Ethernet Switch. https://people.ucsc.edu/~warner/Bufs/PB_Spectrum-2.pdf.

[2]Both papers also include some discussion about offloading functionality to programmable switches. However, the discussion in both papers largely focuses on SmartNICs.

[9] Netronome showcases next-gen intelligent server adapter delivering 20x ovs performance at open networking summit 2015. https://netronome.com/netronome-showcases-next-gen-intelligent-server-adapter\-delivering-20x-ovs-performance-at-open-networking\-summit-2015/.

[10] Network Virtualization using Generic Routing Encapsulation. https://msdn.microsoft.com/en-us/library/windows/hardware/dn144775%28v=vs.85%29.aspx.

[11] SYN cookies. https://en.wikipedia.org/wiki/SYN_cookies.

[12] The CAIDA UCSD Anonymized Internet Traces 2016 - April. http://www.caida.org/data/passive/passive_2016_dataset.xml.

[13] Xilinx smart network interface card. https://www.xilinx.com/applications/data-center/smart-network-interface-card.html.

[14] XPliant™ Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

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

[16] 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.

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

[18] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker. HotCocoa: Hardware Congestion Control Abstractions. In *HotNets*, 2017.

[19] Arpit Gupta and Rob Harrison and Marco Canini and Nick Feamster and Jennifer Rexford and Walter Willinger. Sonata: Query-Driven Streaming Network Telemetry . In *SIGCOMM*, 2018.

[20] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. *ACM International Measurement Conference*, Nov. 2010.

[21] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.

[22] R. Caceres, N. G. Duffield, J. Horowitz, and D. F. Towsley. Multicast-based inference of network-internal loss characteristics. *IEEE Transactions on Information Theory*, Nov 1999.

[23] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-Based Congestion Control. *Queue*, Oct. 2016.

[24] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *HotSDN*, 2012.

[25] A. Caulfied, P. Costa, and M. Ghobadi. Beyond SmartNICs: Towards a Fully Programmable Cloud. In *HPSR*, 2018.

[26] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016.

[27] M. Dalton et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI*, 2018.

[28] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[29] D. Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *NSDI*, 2017.

[30] D. Firestone et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[31] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014.

[32] Y. Geng, S. Liu, F. Wang, Z. Yin, B. Prabhakar, and M. Rosenblum. Self-programming networks: Architecture and algorithms. In *Allerton*, 2017.

[33] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*, 2015.

[34] V. Jacobson and M. J. Karels. Congestion Avoidance and Control. In *SIGCOMM 1988*, 1988.

[35] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*, 2018.

[36] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*, 2017.

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

[38] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*, 2017.

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

[40] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM Industrial Demo Session*, 2015.

[41] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *SIGCOMM*, 2018.

[42] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[43] B. Li, K. Tan, L. Luo, R. Luo, Y. Peng, N. Xu, Y. Xiong, and P. Cheng. ClickNP: Highly Flexible and High-performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, 2016.

[44] Y. Li, D. Wei, X. Chen, Z. Song, R. Wu, Y. Li, X. Jin, and W. Xu. DumbNet: A Smart Data Center Network Fabric with Dumb Switches. In *EuroSys*, 2018.

[45] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*, 2016.

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

[47] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

[48] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM*, 2016.

[49] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*, 2017.

[50] Nofel Yaseen and John Sonchack and Vincent Liu. Synchronized Network Snapshots. In *SIGCOMM*, 2018.

[51] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *OSDI*, 2016.

[52] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *OSDI*, 2018.

[53] F. L. Presti, N. G. Duffield, J. Horowitz, and D. Towsley. Multicast-based Inference of Network-internal Delay Distributions. *IEEE/ACM Transactions on Networking*, Dec. 2002.

[54] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *NSDI*, 2017.

[55] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*, 2018.

[56] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *SIGCOMM*, 2012.

[57] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *SIGCOMM*, 2015.

[58] 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.

[59] 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.

[60] I. Stoica, S. Shenker, and H. Zhang. Core-stateless Fair Queueing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High-speed Networks. *IEEE/ACM Transactions on Networking*, 2003.

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

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

[63] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution Measurement of Data Center Microbursts. In *IMC*, 2017.

[64] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *SIGCOMM*, 2015.