When Should The Network Be The Computer?

Dan R. K. Ports
dan@drkp.net
Microsoft Research
Redmond, WA

Jacob Nelson
Jacob.Nelson@microsoft.com
Microsoft Research
Redmond, WA

Abstract

Researchers have repurposed programmable network devices to place small amounts of application computation in the network, sometimes yielding orders-of-magnitude performance gains. At the same time, effectively using these devices requires careful use of limited resources and managing deployment challenges.

This paper provides a framework for principled use of in-network processing. We provide a set of guidelines for building robust and deployable in-network primives, along with a taxonomy to help identify which applications can benefit from in-network processing and what types of devices they should use.

CCS Concepts • **Networks → In-network processing**; *Data center networks*; • **Software and its engineering → Distributed systems organizing principles**.

Keywords in-network computation, reconfigurable devices, programmable switches, smart NICs

ACM Reference Format:
Dan R. K. Ports and Jacob Nelson. 2019. When Should The Network Be The Computer?. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321439

1 Introduction

Datacenters are no longer limited to commodity hardware. The end of Moore's law and economies of scale in the largest datacenters make deploying custom architectures a realistic option. Reconfigurable devices are being deployed at large scale – notably in the network path, where programmable switches [5, 7, 49], FPGAs [11, 31], and other smart NICs [8, 34] are available. These devices allow a limited form of *in-network computing*: application-specific functions can be run in hardware at line rate, offering orders of magnitude higher throughput and lower latency than can be achieved by a traditional server.

Given the performance gains possible with in-network computing, it's no surprise that researchers have explored a variety of new applications in recent years. Traditional networking applications, such as congestion control [40, 41], load balancing [20, 33], and packet scheduling [42] are only the beginning. In-network computing can be used to implement "systems" functionality such as consensus protocols [9, 10, 27, 36], concurrency control [17, 26], aggregation primitives [32, 38, 39], and query processing operators [13, 25]. Other work offloads entire applications to programmable devices, including key-value stores [46], network protocols like DNS [43], and even industrial feedback control [37].

Thus far, in-network computing research has studied particular applications and how they might be implemented in network devices. We believe it is time to take a step back and examine not just what in-network computing *can* do but what it *should* be used for. Towards this end, we aim to provide a principled approach to the question of when to use in-network computing and how to deploy it. Specifically, in this paper:

- We review the benefits and limitations of today's major programmable network platforms (§2) and how they can be deployed (§3).
- We argue for a deployment approach based on a library of reusable network primitives (§4).
- We provide a taxonomy that classifies which applications show promise for in-network computing, based on classifying their computation, state, and communication requirements (§5).
- We discuss several open research challenges that must be addressed for in-network computing to have practical impact (§6).

2 Technology Background

In-network computing is enabled by hardware platforms that provide flexible, line-rate data processing capabilities on the network path. We begin with a brief overview of the platforms available today.

Programmable dataplane switches. A recent generation of switch ASICs provides programmability on the data path.

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321439

These go beyond prior software-defined networking by allowing custom logic to run at per-packet granularity rather than on the level of flows. Fully-programmable architectures like Barefoot's Tofino [5] and Cavium's XPliant [49] have attracted the most attention, but programmability is also being integrated into traditional switch architectures. For example, Broadcom's Trident3 line [7] offers five programmable stages in addition to the conventional fixed-function pipeline.

Many of the architectural details differ between specific platforms, but in general they follow a reconfigurable match-action model [6]. Packets pass through a pipeline of 10-20 stages, each using a longest-prefix *match* against a set of rules to select an associated *action*. Such an action might consist of a set of simple ALU operations, accessing a small amount of memory, that can modify the packet or its destination port(s).

FPGA network devices. Attaching one or more Ethernet interfaces to a general-purpose FPGA provides a versatile architecture. Smart NICs based on this architecture have been widely deployed by major cloud providers [2, 11], and similar devices are commercially available [31, 50]. Although these are NIC designs, in all cases the programmability is entirely on the network side, i.e., the host-to-NIC interface remains unchanged. The same approach has been extended to network switches, though achieving line-rate performance for many high-speed interfaces remains a challenge: few boards available today support more than 4 ports at > 10 Gb speed; the largest FPGAs available support only 20 100 Gb Ethernet links [16, 48]. Hybrid architectures use a conventional switch ASIC for the normal case, while routing specially-selected packets to a FPGA for further processing [3].

Multicore network processors. Other programmable NICs use many general-purpose CPUs for packet processing [8]. These provide more computational power and are easier to program, though scalability to high throughput is a challenge: this architecture can't be used with switches, and even supporting 100/400 Gb Ethernet is questionable [11].

2.1 The Bottom Line

Despite different architectures and programming models, several key properties remain constant across in-network computing platforms:

Line-rate processing with minimal latency. These devices are designed to run at the full speed of their interfaces – with total bandwidth as high as 12.8 Tbit/s for programmable switches. Programmable switch pipelines achieve similar latency to conventional switches, and even programmable NICs have much lower latency than end hosts because placing the computation close to the network avoids PCIe bus- and OS-induced latency.

Throughput vs computation. A general tradeoff exists between total network bandwidth and available computation per packet. Programmable switches are limited to a handful of simple arithmetic or logic operations (not even integer multiplication [40]). FPGAs provide a more flexible tradeoff, but increasing the IO bandwidth reduces available gates for computation, and multicore SoCs offer the most general computation but with limited throughput.

Throughput vs storage. Similarly, storage is limited in high-bandwidth devices. Programmable switches are restricted to ~ 10 MB on-die SRAM. FPGAs have a similar amount, but can also be configured to use external DRAM at significantly lower bandwidth. Persistent storage, if available, is not fast enough for line-rate access.

We do not foresee either of these tradeoffs changing dramatically, as they are tied to fundamental architecture constraints. As long as devices aim to meet line-rate performance requirements, computation and storage must be on-die and meet strict timing requirements.

3 Deployment Approaches

We envision two basic deployment options for in-network computing devices:

In-fabric deployment is "true" in-network computing: computation is placed on the network path, using programmable switches or NICs in the place of conventional ones. This achieves two of the key benefits of in-network computing: (1) all traffic flowing through a switch or NIC is captured, potentially giving a global view of traffic to a rack or cluster, and (2) there is no additional latency to route packets to the device. The main tradeoff is that the limited computation and memory resources become even more constrained, as it must support conventional routing and forwarding logic as well.

End-device deployment is an alternative where in-network processing devices are used, but *separately* from the fabric. Rather, they are attached to the network as a normal device – in essence, a specialized accelerator. This doesn't provide the same latency benefits as an in-fabric deployment, though it can still provide higher throughput and lower latency than a conventional server. The major benefit is incremental availability: a few devices can be deployed without rearchitecting the entire datacenter network.

In the long term, the in-fabric approach offers the greatest computational and latency benefits. It does not require additional hardware, beyond the availability of programmable network devices; experience has shown that this already provides a measurable benefit for FPGA NICs. In the interim, we expect to see end-device deployments first, offering a way to experiment with programmable hardware for specific applications with less deployment and management overhead. However, in addition to the additional cost of dedicated

hardware devices, datacenter operators are increasingly under pressure to reduce the number of specialized rack- and cluster-level hardware configurations they support, in order to ease scheduling and deployment.

4 Principles for In-Network Computation

While in-network devices are capable of computation, they are far from general-purpose computers. Both the architectural limitations described in §2 and the need to co-locate computation and existing network functionality on the same device for an in-fabric deployment mean that the computation that can be done on each packet is highly restricted.

In this respect, building a system with in-network computation is quite different than conventional system implementation. In the same way, it differs from the line of active networks research two decades ago that also proposed moving computation to the network [44]: that work used general-purpose CPUs for routing [1, 45], allowing far more complex in-network logic. Revisiting this approach today requires using hardware that can meet the rapidly-increasing speeds of the datacenter network.

We propose the following principles for effectively using in-network computation:

Offload primitives, not applications. It is possible, in some cases, to offload entire applications to programmable network devices. In most cases, though, implementing a full application-level protocol requires too much computation or storage to be practical. Instead, designers should identify a key, common-case primitive from the application that can be implemented in the network, and leave the rest to a conventional application-level implementation. This is an example of co-designing the application with the network [36].

We believe this is an important principle both for programmable switches and NICs. It's tempting to think that strict resource limits apply only to programmable switches, and that it's possible to offload more functionality to a programmable NIC. However, experience with programmable NICs currently being used to implement network virtualization suggest that FPGA space is a scarce resource: any available space is quickly snatched up by different groups wanting to implement new functionality.

Make primitives reusable. Beyond supporting devices with limited computational capacity, the offload-primitives approach also makes system development more practical. Programming for a FPGA or programmable switch requires considerably more development effort, arguing for leaving most application logic in regular software. And keeping the in-network functionality small eliminates the need for it to be updated regularly, a complex deployment problem.

These benefits are magnified when primitives are reusable across different applications, suggesting there is much value in identifying a library of reusable primitives with standard interfaces. While prior work is highly application-specific, there is some evidence that primitives may be reusable: timestamping-like primitives have been used not just for consensus [27] and transactions [26], but also network debugging [4], and in-network aggregation has been proposed for DNN training [38, 39], database queries [25], and streaming network telemetry [13]. Identifying application-independent primitives leads naturally into a standardization process, and early IRTF efforts are ongoing [15].

Preserve fate sharing. Network devices fail. Their failure should not render unavailable a system that is using them for computation. This means that systems must be able to recover from switch or NIC failure. An exception is when the device already represents a single point of failure for a set of nodes, e.g., a NIC for one server or a top-of-rack switch.

Keep state out of the network. A key way to achieve fault tolerance is to maintain only soft state in network devices; in the event of their failure, an application-level protocol can recover the lost data from servers.

Minimal interference. New primitives should first do no harm to existing network functionality. For example, they should not interfere with policies for how packets are routed unless absolutely necessary. This reduces the potential impact to network reliability.

5 Classifying Potential Benefits

To help choose the right primitives to run in the network, and the right platform and deployment model, it is valuable to be able to estimate the benefit that could be obtained by using in-network computing for a primitive. To this end, we propose a taxonomy of in-network computing applications. The goal is to classify primitives by whether they will be compute-, memory-, and/or bandwidth-bound; with that information, decisions about platform and deployment model can be made.

We classify applications along three axes. For each axis, we primarily consider the asymptotic behavior of the application. In the applications we studied, the variable is often the packet size n, the number of replicas r, or the application working set size s. However, both asymptotic and actual values matter: an application with $O(n^2)$ behavior may fit only if n is small, and an application with $O(s)$ operations per packet may be impractical if s is too large. The asymptotic values provide a starting point for further investigation. The three axes are:

Operations per packet. How many operations must be performed to process a received packet? Most applications we examined either execute $O(1)$ operations per packet (the minimum), or $O(n)$ operations, where n is the packet length. Applications with $> O(n)$ operations will be challenging to run on in-network computation devices. As a shorthand, we

classify applications along this axis as C (constant), L (linear), or G (greater than linear).

State per packet. How much data is required to be resident in each in-network computing device to process a packet? For many applications, only a fraction of this data may be touched while processing each packet, but which data to access may not be known until the packet is received. We see three common cases here: $O(1)$ state (the minimum), $O(n)$ state, or $O(s)$ state, where s is the application working set size. Again, we classify applications along this axis as C (constant), L (linear), or G (greater than linear).

Packet gain. For each packet received by an in-network computing device, how many packets are sent? We see three primary cases here: those with $O(1)$ behavior that emit the same number of packets as they consume; those with $O(r)$ behavior that broadcast their output to r replicas, and those with $O(1/r)$ behavior, which emit a single aggregate result from data collected from r replicas. On this axis, we classify applications as – (less than constant), C (constant), or + (greater than constant).

Classifying applications. We label an application by combining the three axes: a system with constant operations per packet, linear state, and high packet gain is labeled CL+. Once an application has been classified, the dominant axis is determined, first by considering asymptotic behavior, and then by comparing actual values if the asymptotic comparison is ambiguous. We give precedence to packet gain when possible: if an application has non-constant packet gain, it can obtain significant benefits from running in a switch due to its high degree of connectivity, but only if its state and compute limits are compatible with the switch's limitations. Otherwise, the dominant axis is either operation count or state.

Table 1 shows our estimates of the behavior of a number of recently-published in-network computing systems according to our taxonomy. We see some common themes: first, most systems have $O(1)$ operations per packet, and only one has more than the packet size. This is to be expected, since these systems have already been adapted for in-network operation. Second, most systems have linear state, but the magnitude of that state varies significantly; linear state is not a guarantee of implementability. We now explain the classification of a few applications.

Network sequencing [26, 27] accelerates replication through an in-network sequencing primitive. This is a CC+, gain-dominated application in our taxonomy. Since computation and state requirements are minimal, in-switch implementation is ideal.

Cloud providers accelerate virtual networks by offloading policy and filtering to hardware with a lookup table for active flows as primitive. This is a CLL, state-dominated application, and is best suited for middlebox instead of a switch, for two reasons: First, the packet gain is 1, so a switch provides no throughput benefit. Second, while the state per packet is linear in the flow table size, the actual flow table size is large (> 1 million flows [11]). This is practical only in an FPGA or NPU with external DRAM. Azure SmartNIC [11] uses this design.

In-network storage is a CLC application, and – not surprisingly – state-dominated. Our taxonomy suggests that it therefore may not be well-suited to programmable switches. At first glance, this stands in contrast to recent efforts that have shown that programmable switch implementations are possible [18, 19]. However, these are specifically limited to restricted applications (e.g., small caches [19]) for the same reason. We believe that the resource limits will prove even more restrictive in production environments, making designs that use FPGAs with external DRAM [46] or maintain only metadata in the switch [28, 29] a more practical alternative.

Deep neural network training can be accelerated with network support [30, 38, 39] uses a primitive that sums vectors from multiple replicas in network devices rather than in a parameter server. (Database reductions can use a similar primitive [25].) This is a LL–, gain-dominated application due to the reduction in communication with the parameter server. Since the packet gain is less than 1, implementation in a switch will provide the most benefit as long as the computation and state requirements can be satisfied. Both of these are linear in the size of the packet, so a switch implementation should be possible (and is ongoing work).

In-network inference of deep neural networks, on the other hand, actually does the entire computation in the device. This is a GLC application whose dominant axis is computation; one implementation [12] does $O(d^2)$ operations for inputs of dimension d for a long short-term memory (LSTM) network. In terms of actual values this is 64 million operations and 32MB of state. Clearly this would not fit in a switch; it fits in an FPGA NIC only by using special narrow-precision numeric representations.

6 Open Challenges

Beyond the question of which applications are a good fit, there are many challenges that must be addressed to deploy in-network computation in a large-scale, realistic environment. Some of these have not received much attention in the research community; we aim to shine light on them here.

Scale and decentralization. Datacenter networks contain thousands of switches. Many prior systems (including some of our own work!) have assumed that it is feasible to ensure that a single switch handles all traffic destined for a particular application. This may be practical for certain small-scale systems, e.g., a rack-scale storage system or a dedicated ML training cluster. In general, however, it requires restricting either the placement of servers to specific racks within a

In-network primitive	Ops/packet	State/packet	Packet gain	Class	Dominant
Network sequencing [26, 27]	$O(1)$	$O(1)$	$O(\|\text{replicas}\|)$	CC+	Gain
Replicated storage [18]	$O(1)$	$O(\|\text{dataset size}\|)$	$O(1)$	CLC	State
Caching [19, 29]	$O(1)$	$O(\ln(\|\text{dataset size}\|))$	$O(1)$	CLC	State
DNN training (allreduce) [30, 38, 39]	$O(\|\text{packet}\|)$	$O(\|\text{packet}\|)$	$O(1/\|\text{replicas}\|)$	LL-	Gain
DNN inference [12]	$O(\|\text{input size}\|^2)$	$O(\|\text{model size}\|)$	$O(1)$	GLC	Ops
Database reductions [25]	$O(\|\text{packet}\|)$	$O(\|\text{elements}\|)$	$O(1/\|\text{replicas}\|)$	LL-	Gain
Database hash joins [25]	$O(1)$	$O(\|\text{elements}\|)$	$< O(1)$	CL-	State
Virtual networking [11]	$O(1)$	$O(\|\text{flow table}\|)$	$O(1)$	CLC	State
In-band network telemetry [22]	$O(1)$	$O(1)$	$O(1)$	CCC	Ops

Table 1. Classifying recently-published in-networking computing primitives according to our taxonomy.

datacenter or the network routes taken to access them. Both are barriers to adoption.

Scalable systems that treat the network not just as one switch but as a distributed system of switches are needed. Many of the techniques needed have already been explored, e.g., replicating data across switches [10, 18] or accessing remote device memory [23]. We suggest using these not as applications in their own right but as abstractions to build other in-network applications.

In-device parallelism. Network processors often have parallel pipelines with restricted communication between them – a programmable switch might have 4 pipelines of 16 ports each [5]. Many systems have not been tested at enough scale to require multiple pipelines, and the lack of inter-pipeline shared state may pose problems. This is a restricted case of the multi-device parallelism discussed above. However, it may have simpler or more efficient solutions, e.g., carefully selecting the output pipeline or recirculating packets to a new input pipeline in order to ensure that the correct computational state is available.

Logical vs wire messages. Network devices do their processing at the packet level. This doesn't always correspond to the logical requests that a system operates at – indeed, the TCP stream abstraction specifically obscures this. Despite the ubiquity of TCP applications, most in-network computation systems today work exclusively with UDP messages. This semantic gap must be bridged.

Encryption. Network devices can't process data that they can't read, and network traffic is increasingly encrypted, even within a datacenter. Two things give us hope that this problem is solvable. First, many primitives operate on metadata that is not as sensitive, e.g., assigning sequence numbers to packets, and (with careful protocol design) could be left unencrypted. Second, homomorphic encryption techniques allow computation on encrypted data. While these are infeasibly expensive for general functions, the types of computation done in the network are by necessity simple, restricted ones – e.g., summing integers for in-network aggregation, or counting packets for in-network telemetry – lending tractability to the problem. Work on privacy-preserving middleboxes has demonstrated feasibility for some applications [24].

Multitenancy. If multiple applications are running on the same network device, how will contention be resolved? This question has largely remained unaddressed – perhaps because the killer apps for this functionality have not yet emerged. Much like the earliest computers, today's programmable devices effectively run on the data plane a single monolithic program that assumes unfettered access to all hardware resources. Classic operating system abstractions like virtualization, resource management, and protection domains are needed. Despite the "operating system" term, we imagine that this would in fact be achieved primarily by compile-time mechanisms that implement resource partitioning (cf. [21]).

Isolation. Looking further, we may also wish to support mutually-distrusting tenants, e.g., to allow cloud customers to deploy their own primitives. This poses obvious security challenges – not just between the different tenants, but also the network operator, who will be understandably reluctant to allow untrusted code to run on their infrastructure. Initially, such isolation will likely be achieved by running customer code on separate devices – as supported by the end-device deployment model. Safely consolidating these onto a single device will require defining a security model that specifies which traffic a given client can view and how it can be manipulated. This model must then be enforced on client-provided programs, ideally using formal methods to provide provable non-interference properties. In the future, manufacturers could include hardware isolation features in their designs: for instance, reserved pipeline stages and memory, or hardware-enclave-like functionality, as proposed by recent research [14, 35, 47].

Interoperability. Large-scale deployments need to support multiple hardware platforms: single-vendor solutions are out of the question for cost and supply chain reasons. High-level languages like P4, NPL, or PX aim to provide device independence – but the reality is quite different. Even setting aside that different vendors offer conflicting standards, what

compiles in P4 on one device may not be the same as on another with different hardware resources. In the future, the existence of conflicting software on the same device, as described above, will exacerbate this challenge. Recent research in optimizers and program synthesis tools may help make true device-independence more of a reality.

7 Conclusion

Offloading application-level computation to programmable network devices presents major design and deployment challenges, but also offers impressive end-to-end performance gains. We argue that these benefits are best extracted via a library of reusable network primitives. Our taxonomy provides a guide to which primitives will have the most value: those that require little per-packet state or computation, or those that can reduce network utilization.

Acknowledgments

We thank Ricardo Bianchini, Xin Jin, Arvind Krishnamurthy, Michael Lowell Roberts, Mothy Roscoe, Lihua Yuan, and Irene Zhang, along with the anonymous reviewers, for their helpful feedback.

References

[1] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):29–36, July 1998.

[2] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/.

[3] Arista 7124FX Application Switch. https://www.arista.com/assets/data/pdf/7124FX/7124FX_Data_Sheet.pdf.

[4] Arista latency analyzer (LANZ). https://www.arista.com/assets/data/pdf/TechBulletins/Lanz.pdf.

[5] Barefoot Networks. Tofino. https://barefootnetworks.com/products/brief-tofino/.

[6] 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 *Proceedings of ACM SIGCOMM 2013*, Hong Kong, China, Aug. 2013. ACM.

[7] Broadcom Trident3. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series/.

[8] Cavium LiquidIO II. http://www.cavium.com/LiquidIO-II_Network_Appliance_Adapters.html.

[9] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. Network hardware-accelerated consensus. Technical Report USI-INF-TR-2016-03, Università della Svizzera italiana, May 2016.

[10] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, H. Weatherspoon, M. Canini, N. Zilberman, F. Pedone, and R. Soulé. P4xos: Consensus as a network service. Technical Report USI-INF-TR-2018-01, Università della Svizzera italiana, May 2018.

[11] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 51–66, 2018.

[12] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press.

[13] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of ACM SIGCOMM 2018*, Budapest, Hungary, Aug. 2018. ACM.

[14] J. Han, S. Kim, J. Ha, and D. Han. SGX-Box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, pages 99–105, Hong Kong, China, 2017. ACM.

[15] J. He and R. Chen. In-network data-center computing. Internet Draft draft-he-coin-datacenter-00, IRTF, Oct. 2018. https://datatracker.ietf.org/doc/draft-he-coin-datacenter/.

[16] Intel. Intel Stratix 10 product table. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-tx-product-table.pdf.

[17] T. Jepsen, L. P. de Sousa, M. Moshref, F. Pedone, and R. Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the ACM SIGCOMM 2018 Workshop on In-Network Computing (NetCompute '18)*, Budapest, Hungary, Aug. 2018. ACM.

[18] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, 2018. USENIX Association.

[19] 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 *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Beijing, China, Oct. 2017. ACM.

[20] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the 2016 Symposium on SDN Research (SOSR '16)*, Santa Clara, CA, USA, Mar. 2016. ACM.

[21] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, Carlsbad, CA, 2018. USENIX Association.

[22] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, M. Hira, and B. Davie. In-band network telemetry. https://p4.org/assets/INT-current-spec.pdf, 2016.

[23] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. Generic external memory for switch data planes. In *Proceedings of the 16th Workshop on Hot Topics in Networks (HotNets '18)*, Redmond, WA, USA, Nov. 2018. ACM.

[24] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu. Embark: Securely outsourcing middleboxes to the cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 255–273, Santa Clara, CA, 2016. USENIX Association.

[25] A. Lerner, R. Hussein, and P. Cudré-Mauroux. The case for network accelerated query processing. In *Proceedings of the 9th Conference on Innovative Data Systems Research (CIDR '19)*, Asilomar, CA, USA, Jan. 2019. VLDB / ACM.

[26] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using network multi-sequencing. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Beijing, China, Oct. 2017. ACM.

[27] J. Li, E. Michael, A. Szekeres, N. K. Sharma, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, USA, Nov. 2016. USENIX.

[28] J. Li, J. Nelson, X. Jin, and D. R. K. Ports. Pegasus: Load-aware selective replication with an in-network coherence directory. Technical Report UW-CSE-18-12-01, University of Washington CSE, Seattle, WA, USA, Dec. 2018.

[29] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, Santa Clara, CA, 2016. USENIX Association.

[30] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy. Parameter Hub: A rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 41–54, Carlsbad, CA, USA, 2018. ACM.

[31] Mellanox Innova2. http://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex.

[32] Mellanox scalable hierarchical aggregation and reduction protocol (SHARP). http://www.mellanox.com/page/products_dyn?product_family=261&mtag=sharp.

[33] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of ACM SIGCOMM 2017*, Los Angeles, CA, USA, Aug. 2017. ACM.

[34] Netronome Agilio CX. https://www.netronome.com/products/agilio-cx/.

[35] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. SafeBricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 201–216, Renton, WA, 2018. USENIX Association.

[36] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in datacenter networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, USA, May 2015. USENIX.

[37] J. Rüth, R. Glebke, K. Wehrle, V. Causevic, and S. Hirche. Towards in-network industrial feedback control. In *Proceedings of the ACM SIGCOMM 2018 Workshop on In-Network Computing (NetCompute '18)*, Budapest, Hungary, Aug. 2018. ACM.

[38] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th Workshop on Hot Topics in Networks (HotNets '17)*, Palo Alto, CA, USA, Nov. 2017. ACM.

[39] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtarik. Scaling distributed machine learning with in-network aggregation. Technical report, KAUST, Feb. 2019.

[40] N. K. Sharma, A. Kaufmann, T. Anderson, C. Kim, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, USA, Mar. 2017. USENIX.

[41] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, USA, Apr. 2018. USENIX.

[42] 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 *Proceedings of ACM SIGCOMM 2016*, Florianopolis, Brazil, Aug. 2016. ACM.

[43] N. Sultana, S. Galea, D. Greaves, M. Wójcik, J. Shipton, R. G. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. R. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman. Emu: Rapid prototyping of networking services. In *Proceedings of the 2017 USENIX Annual Technical Conference*, Santa Clara, CA, USA, June 2017. USENIX.

[44] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications*, 35(1):80–86, Jan. 1997.

[45] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *ACM SIGCOMM Computer Communication Review*, 37(5):81–94, Apr. 1996.

[46] Y. Tokusashi, H. Matsutani, and N. Zilberman. LaKe: An energy efficient, low latency, accelerated key-value store, 2018. https://arxiv.org/pdf/1805.11344.pdf.

[47] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 2:1–2:14, Los Angeles, CA, USA, 2018. ACM.

[48] Xilinx. Xilinx UltraScale+ fpgas product tables and product selection guide. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[49] XPliant Ethernet switch product family. www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[50] N. Zilberman, Y. Audzevich, G. Kalogeridou, N. Manihatty-Bojan, J. Zhang, and A. Moore. NetFPGA: Rapid prototyping of networking devices in open source. In *Proceedings of ACM SIGCOMM 2015*, London, United Kingdom, Aug. 2015. ACM.