Infinity: A Scalable Infrastructure for In-Network Applications

***Abstract*—Network programmability is an area of research both defined by its potential and its current limitations. While programmable hardware enables customization of device operation, tailoring processing to finely tuned objectives, limited resources stifle much of the capability and scalability desired for future technologies. Current solutions to overcome these limitations simply shift the problem, temporarily offloading memory needs or processing to other systems while incurring both round-trip time and complexity costs. To overcome these unnecessary costs, we introduce Infinity, a resource disaggregation method to move processing to capable devices while continuing to forward as the original owner, limiting unnecessary buffering and round-trip processing. By forwarding both the processing need and associated data simultaneously we are able to scale operation with minimal overhead and delay, improving both capability and performance objectives for in-network processing.**

I. Introduction

The Internet was designed over 40 years ago, and in the decades that followed, the ubiquity of it both served as a great testament to its architecture, but also introduced new operational challenges. In particular, with new applications and uses there were new requirements, many of which were not supported under the current architecture. While the research community followed with innovative solutions, the ossification served as an impediment to adoption [14]. Today, this problem is even more pronounced with the rise of edge computing and proposals for tactile applications that have the potential to revolutionize health-care, for example. These applications require high availability, security, and low latency that the current Internet cannot provide.

Towards the goal of enabling this future, researchers have turned to technology ranging from virtualization [2], [14] to programmable network hardware [12], [3], [4]. Our focus here is on the programmable network hardware. Traditional networking equipment follows a fixed function design model (e.g., ASICs) typically optimized for a narrowly defined role, such as a datacenter, distribution, or core device. This design approach provides a refined high performance architecture with limited versatility outside predefined operational scopes. However, this limits the degree of potential innovation, requiring operators to implement solutions that align with often out-dated pre-coded characteristics of the network device. This changed with the introduction of the Protocol Independent Switch Architecture (PISA) [4], which provides a hardware design in which the protocol parsing and packet processing are entirely programmable.

What this means for the next-generation Internet is that there are now unique opportunities for in-network computing capabilities at high-performance and low-latency that were not possible before. In recent years, in-network applications have been used to reduce latency for key-value stores [8], support distributed locking [19], and performing data aggregation to accelerate machine learning [15]. These all point to the great potential for in-network computing that comes with these new programmable hardware architectures.

Unfortunately, there's a catch. The architecture is characterized by a pipeline of match-action tables, with, for example, an associate TCAM to perform quick lookups, an ALU to enable flexible actions, and SRAM to enable state storage. Being that it is hardware, there's a limitation on the amount of each that are available. At the same time, innovations seek to do more in-network - both in terms of the complexity of the designs, but also in the number of concurrent applications we want to support. Interestingly, general purpose computing platforms (CPUs) have similar resource limitations, but have established operating system abstractions that mask them (e.g., virtual memory for extended memory, and CPU scheduling for multiple concurrent applications). We argue that programmable hardware needs its own set of abstractions to meet the growing needs of in-network applications and dynamic architectures.

One such related work that sought to overcome the resource constraint around memory on programmable switches, was TEA (for Table Extension Architecture) [11]. For this, they built on recent advances in computing infrastructures that leveraged RDMA capabilities to enable fast remote memory [6], [7]. By using RDMA to a remote server, they illustrated how they could implement enough of the RDMA protocol in a P4-enabled switch to be able to access memory on a remote server. This meant that if they ran out of memory on a switch, they could buffer a packet and do a remote read to a data store, effectively extending the memory available. Unfortunately, this has three key limitations: (i) it only works for memory resources, (ii) it incurs a buffering penalty on the switch, and (iii) it incurs a round-trip time in the middle of the packet processing.

To address these three challenges we propose *Infinity*, a programmable network architecture which abstracts networking hardware into virtual aggregated hardware sets, giving in-network applications the illusion of having infinity processing capacity. Infinity ensures that in-network applications can locate resources within the aggregated hardware sets neces-

sary to meet processing objectives by leveraging data plane dis-aggregation, scale-out techniques (sequential decomposition/vertical scaling and scale-out horizontal scaling), and per-function tailored performance requirements.

With Infinity we enable the following contributions to improve current in-network applications by leveraging programmable networking hardware:

- We provide abstractions for building scalable and flexible in-network applications on top of programmable hardware enabling high-performance and low-latency processing.
- We enable in-network applications to be deployed on top of our *Virtual Infinity Switch* abstraction, giving them the illusion of infinite computation resources.
- We provide a mechanism to fully and efficiently utilize the network infrastructure within the data center.

The remainder of this paper is organized as follows: In section II we frame the problem of fully flexible internet architectures by highlighting limitations with current programmable solutions. We then introduce our architectural solution, Infinity, and demonstrate how we address the limitations of current dis-aggregation and scalability solutions in section III. In section V we propose future directions with Infinity to enable fully flexible architectures while also addressing limitations to our approach. Finally, in section VI we conclude the paper and discuss the next steps in the direction of realizing our vision.

II. Background

A. Programmable Hardware

Modern data centers operate in a highly dynamic environment of competing stakeholder interests. Here, architects are pressured to provide solutions that support both dynamic per-individual use-cases with high performance, efficient, and agile functionality. Competitiveness of a provider is therefore linked to how well they can provide dynamic solutions that are naturally at odds with current static architectures. Two solutions currently exist to address this problem: programmable hardware and network virtualization.

With programmable networking hardware operators can add new functionality within the network, bypassing traditional development pipelines while increasing product lifecycles. Two common examples of programmable hardware include: protocol independent switch architecture (PISA) based switches and programmable SmartNICs. The versatility provided by programmable hardware appeals to the modern data center, enabling tailored network functions to meet performance, flexibility, energy efficiency, or cost effective business objectives [].

To allow for simple configuration, domain specific languages (DSLs) like P4 [3] were introduced to bridge the gap between low-level hardware configuration and operator knowledge. Critically, P4 is vendor agnostic with platform specific compilers provided by all of the common programmable hardware manufacturers []. P4 has constructs capable of programming the different components of a programmable networking device, producing both a flexible and feature complete data plane. For example, with P4, an operator may tailor their hardware to enable custom packet header parsers and de-parsers, or to define matching fields and actions to be included on the match action tables (MATs). It also provides a runtime, that allows defining APIs for data/control plane communication.

B. Resource Limitations with Programmable Hardware

Although both SmartNICs and programmable switches offer significant customization with packet processing capabilities, they have finite resources which impose constraints on in-network processing scalability and desired functionalities. For example, modern programmable switches are limited in fast memory resources (i.e., SRAM) [13], [11] and have a finite amount of processing units (e.g., ALUs) [17] that can be used to compose the processing pipelines. This limitation creates a challenge to run in-network applications with complex logic while simultaneously maintaining high-performance and scalability [18]. Adding to this challenge, diverse programmable hardware sets are built with different underlying architectures and targeted functionalities, making no single type of equipment suitable for running all problem sets [16].

C. Hardware Abstractions

In order to overcome limitations with SRAM availability on programmable switches, a table extension architecture (TEA) [11] was developed to leverage remote memory on RDMA capable commodity servers. Utilizing RDMA effectively works to expand the memory capacity on programmable hardware, making them more suitable to run stateful NFs at scale (e.g., NATs, stateful load balancers, and firewalls). This approach also creates an abstraction layer to extend the switches' MATs (originally stored in local SRAM to ensure high-performance) using remote DRAM, which is accessed directly from the switches data plane whenever an entry cannot be found within the local SRAM. To avoid stalling packet processing during remote look ups, TEA further provides a method in which a triggered packet for the remote look-up is encapsulated in the RDMA request and temporally stored on remote DRAM. When the RDMA server sends the response to the look up, the packet is retrieved along with remote state contents, after which the deferred packet is processed according to the contents retrieved from remote memory. TEA further leverages the RDMA protocol to minimize overheads that could impact its data plane performance. For example, a single set of RDMA state for each queue pair (QP) is maintained based on assumptions about the underlying Ethernet network topology (i.e., Ethernet is considered to be reliable and switches are considered to be directly connected to the RDMA hosts).

D. Challenges with Disaggregation

While TEA can scale hardware resources on a programmable switch through memory dis-aggregation, deferring packet processing during state look ups increases processing inefficiencies on the network. First, packets will suffer from added latency caused by at least one round-trip time (RTT) before they are finally processed. Second, sending the packet to remote data store and querying state on remote memory will cause extra load on the network links. Third, while this approach can increase available memory for the programmable hardware entities, it cannot scale processing resources, as the packets will be returned to the same entity for further processing. A final further limitation of this approach is that TEA leverages precious DRAM resources on remote hosts which could instead support applications to run business logic (e.g., big data, analytics, etc).

III. Introducing Infinity

While programmable network hardware provides exciting avenues for introducing in-network applications, there's an inherent limitation in that the devices themselves are resource-constrained. To enable a new level of performance, scalability and flexibility for in-network applications, while ensuring efficient usage of the underlying infrastructure, we present *Infinity*, a system that can virtualize multiple programmable networking hardware components into virtual aggregated hardware sets, giving applications the illusion of having infinite processing and memory capacity. To do so, with Infinity we introduce a set of networking hardware composing primitives which enable operating system like abstractions to dynamically scale the resource allocations for in-network applications.

Before discussing Infinity, it is useful to understand the typical flow for programmable switches today. Traditionally, in-network applications are written in a domain specific language like P4, which is first compiled and then mapped to finite physical resources on a switch based on an architecture description of the target. This target description will indicate the amount of SRAM, number of ALUs, etc. available. Within the P4 program, the developer's program has an impact on the overall resource usage - e.g., it has to specify the size of each table statically at compile time. If a P4 program is successfully compiled for a given hardware target, it can run at line rate, and the compiler will use resources as needed within the constraints of the target.

In contrast, with Infinity, we aim to perform more of the resource mapping at run-time, such that we can provide the abstraction of an unlimited set of resources. Figure 1 shows a high level representation of Infinity's design. Infinity introduces a compiler that has as a *Virtual Infinity Switch* (VIS) as a target. The compiler is responsible for generating a minimal design for a given in-network application that can be mapped into a VIS. By minimal design, we mean the smallest unit that could be instantiated - e.g., continuing the example above of having to specify the table sizes, we can now assume a reasonable table size and know that it can be expanded later. Further, the VIS abstraction enables an in-network application to expand the resource boundaries of a single switch, ensuring that the developer does not need to care about constraining the processing logic to fit into a single bounded entity.

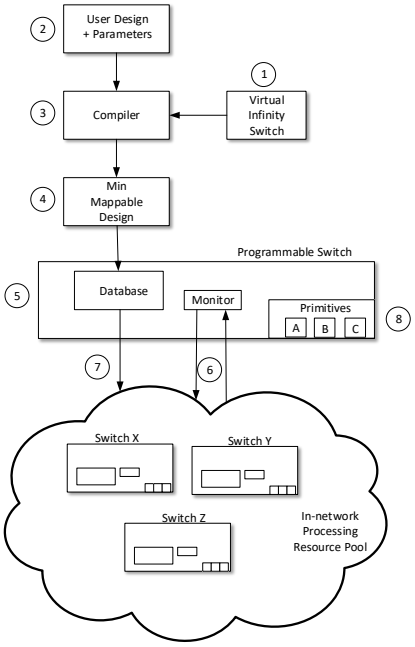

Fig. 1: **Infinity - High Level Deployment** (1), (2) Infinitiy virtual switch architecture is integrated with custom user defined parameters via P4 Language. (3) Compiler prepares system resulting in minimal mappable architecture and loaded onto programmable switch (4). (5) Switch runs Infinity architecture collecting in-network resource metrics from participating systems (6). When action requires resources beyond capability of host, Infinity references resource pool and employs one of its scaling primitives as needed (7).

With a viable minimally mapped design the operator can, at load time, provide that design to the Infinity controller which will in turn place the application onto available resources. Infinity will then continue to monitor the deployed in-network applications and identify bottlenecks, such as SRAM or TCAM exhaustion, due to too many flows being processed. Once a bottleneck is identified, Infinity will employ one of its primitives to expand available resources for the in-network application, as we will demonstrate next.

A. Virtual Infinity Switch

Infinity's main goal is to seamlessly extend resources for running high-demand in-network applications. In this direction, we extend the concepts of the Single Big Router (applied to routers) [10] and Single Big Switch (applied to OpenVswitch) [5], [1] to introduce VIS, an idealized abstraction of a programmable data plane with infinite resources (infinitely many pipelines, infinitely many stages, infinite memory, etc.).

After compilation, in-network application are mapped to a pipeline of programmable forwarding engine stages (or other processing elements) that implement the desired logic. Typically, these map to a single device, which is resource constrained and may prevent applications from being deployed if it's processing requirements go beyond the capacity of the device. This limitation may also prevent in-network applications from continuing to operate when facing resource exhaustion.

To enable seeing a set of switches as single entities, each physical switch inside a VIS has base functionality that supports an overlay that can directly forward to a destination switch. The overlays are built on top of light weight forwarding logic on each switch (i.e., custom header parsing, lookup on a given ID, and forward). This enables flexible composition of primitives that support scaling hardware resources to meet application demands, as we will see next.

Each switch is also assumed to be partially reconfigurable. That is, the switch does not require the entire programmable hardware to be flashed entirely, but instead can be done partially. While this is not entirely supported on some hardware targets for P4, it is supported on FPGA targets which do support partial reconfiguration, and it is inevitable (in our opinion) for ASIC based switches that support P4 as targets.

B. Infinity Primitives

With the VIS abstraction, the underlying network of switches become a pool of resources for a controller to optimize the mapping. To do so, the VIS abstraction needs a collection of primitives that support a dynamically scaling a design. For this, Infinity provides a set of primitives to enable meeting demands with different requirements. The primitives are designed to enable flexible composition of hardware elements allowing scaling resources for in-network applications as needed.

Primitive 1 - Sequential Decomposition:

The first primitive enables a processing pipeline to be split at any point and implemented across two or more switches. This enables applications to increase processing stages, allowing for logic that expands the processing capabilities of a single switch.

To realize this, we need a mechanism to connect the segments of the pipeline spanning the switches. Figure 2 shows how Infinity realizes this primitive. Each switch on the pipeline processes a given packet and at the point the cut was made, Infinity will insert logic to encapsulate it with a custom tag that simply has an ID of the switch where the next segment is mapped to. This is where the pre-configured overlay comes into play - each switch has forwarding logic that can lookup, based on that tag, and forward the packet. When the packet reaches the target switch, the packet is de-encapsulated and processed at the next segment.

Primitive 2 - Horizontal Scaling:

The second primitive in Infinity is horizontal scaling. Much like its counterpart in computing (where there can be replica instances of an application), this primitive enables scaling-out resources in order to increase bandwidth or the number of supported flows (for example) for a given application. Horizontal scaling is allowed at two levels: at a pipeline level or at a pipeline segment level.

To realize this, we need to insert extra logic to connect to the replicas, as shown in Figure 3. First a pipeline (or segment) is replicated to another switch. Then, a load balancing element in the pipeline appends the segment preceding the horizontally scaled segment. If it's the first segment or entire pipeline, this would be at the entry point of the network. One complicating factor is that Infinity needs to know how to split the traffic - what traffic should go to each replica. For this, the application needs to specify a key upon which to partition any memory resources. For example, the key can be the 5-tuple of a flow to ensure all packets for the same flow are processed by the same replica. But, a 5-tuple is not the only way to partition traffic, so we leave this to the application.

A second complicating factor is that the controller needs to know when resources are being exhausted. For logic resources, this can be determined by the bandwidth, but for memory, this is a bit more challenging. To support this awareness, we require the applications to insert an extra bit into tables they want to be expanded. This bit would represent marking space as 'used' or 'unused'. For example, in a dynamic NAT, as new entries are added, they will set the bit to used, and when they time-out or the flow ends, the bit will be set to unused. This allows us to monitor for resource exhaustion.

Primitive 3 - Vertical Scaling:

As the final primitive, Infinity supports vertical scaling. In contrast to horizontal scaling, which adds extra replicas, vertical scaling makes individual instances (which don't have to be replicas) bigger - we allocate more resources to it. As an analogy, consider a virtual machine with 1GB of memory allocated to it. Horizontal scaling will launch an identical virtual machine with 1GB of memory, whereas Vertical scaling will change the allocation of the virtual machine to be 2GB. Vertical scaling can be desirable, for example, in a case where it is not possible to use the horizontal scaling primitive, because it is not possible to use keys to efficiently balance traffic among replicated instances, or if there is no available switch resources to replicate a pipeline or a segment.

There are two ways this can be realized in programmable switches. This first is through disaggregation. This is the mechanism introduced in TEA, where as the switch runs out of memory resources, it can buffer the packet, send a lookup request to remote memory, and then when the lookup returns, it will continue processing as if it got that state from local memory. This is practical due to the speed of RDMA, and through keeping more used state within local memory.

A second mechanism is through migration. As an example, if the application cannot afford the performance penalty and we desire all memory to be local, but local resource constraints cannot fulfill the requirement, we must move it

to a different switch (with a larger allocation). This would then require copying over the state as part of the migration, which can be done in a live manner [9].

C. Orchestrating Infinity

To enable its operation, Infinity relies on a compiler, which maps how physical resources are allocated for a given in-network application. Infinity also relies on a controller with a global view and influence over the network, enabling dynamic resource scaling by leveraging the Infinity primitives.

Infinity Compiler: As we can see in figure 1 Infinity provides a target model with an abstract description of the virtual infinite switch (VIS). This provides information regarding the type of hardware elements that compose the VIS, such that the compiler can map to it. But, as mentioned, as resources are dynamically expandable, the output of the compilation is a minimally mapped design that consists of the unit of deployment from which the application can be expanded at run-time.

Infinity Controller: It is the job of the Infinity controller to find free resources within the pool of network switches and generate a mapping which can be deployed on the physical target. To do this, once the minimally mapped design is placed (allocated to a switch and loaded), the Infinity Controller monitors the current level of utilization for the critical hardware elements that may impact performance. To detect hot spots, Infinity leverages telemetry capabilities on switches and also functionalities implemented on the data plane. For example, as mentioned, SRAM usage on each hardware element is controlled by having each application to update a usage flag once a new flow is added. If a hot spot is detected the controller will work to mitigate it by using one or more of the scaling primitives described in III-B. For example, if an application runs out of SRAM it can leverage the horizontal scaling primitive to increase capacity and process new flows. In this case the controller will replicate the affected pipeline on another switch with enough available resources, and will update the load-balancing rules, ensuring traffic is split among the parallel instances based on a given partition key (e.g., 5-tuple). This enables Infinity to act on a feedback loop, ensuring that applications will meet defined *service level objectives* (SLOs).

IV. Use Cases

Pulling it all together, in this section we illustrate a couple of simple examples that illustrate how Infinity could be used to deploy in-network application that can scale to automatically expand the resources per the application needs.

Layer 4 Load Balancer: A layer 4 load balancer's main purpose is to serve as an entry point for a scalable service, directing traffic to different servers to efficiently distribute the load on each. One of the important characteristics in load distribution is awareness for flow affinity, such that all traffic in a TCP session, for example, is directed to the same server. As such, the load balancer, which can be implemented in modern, programmable switches, needs to store state on the

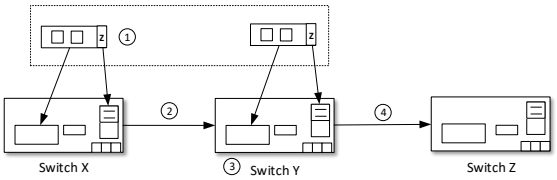

Fig. 2: **Infinity - Sequential Decomposition Operation** (1) The controller determines to sequentially decompose the pipeline and place the first segment on Switch X. The controller sends to Switch X that packets are to be encapsulated with the target header and forwarded according to the lookup table that was built as the overlay (2). (3) Intermediate devices forwards according to packet header until it reaches in-network processing host (4), which decapsulates and processes with available resources.

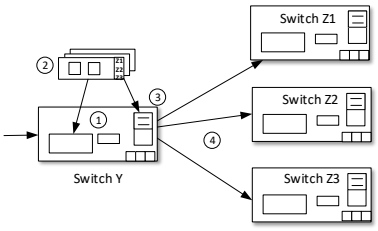

Fig. 3: **Infinity - Horizontal Scaling Operation** The Infinity controller determines there's a need to horizontally scale, placing replicas on switch Z1, Z2, and Z3. (1)Extra logic is added to Switch Y (the preceding segment), which will load balance across these three. (2)The controller sends the switch ID for Z1, Z2, Z3 to Switch Y. (3)This forms the encapsulation that happens at Switch Y, and then packets are forwarded towards the replicas (4). When the packet arrives at a replica, it decapsulates the tag and processes the packet.

switch to remember which server was chosen for each flow. This means that the amount of SRAM allocated is the limiting resource - if you allocate too much, you're wasting valuable resources on the switch, allocate too little and you may not be able to support enough flows. With Infinity, this concern is alleviated with the primitives to scale out. Here, lets say we end up using horizontal scaling. In this case, the 5-tuple would be specified as the partitioning key, and there would be extra logic that splits traffic between two switches to perform the layer 4 load balancing service.

Access Control Lists: An access control list is a set of rules that define whether certain traffic should be permitted or denied passage beyond this particular network point. It would consist of some classification and filtering logic that's coupled to a table with the set of rules. The rules would be added dynamically by a control or management plane application. With this, memory is again, a limiting resource, and directly determines how many rules that could be supported. Ideally, we want to support as many as the operator wishes. As such, we need scaling. If we assume vertical scaling, perhaps because there's wild carding in the rule sets, then there is no clear way to partition traffic. This is ok since these lists are likely change infrequently. In this case, when we detect that the process is running out of memory

on the switch, we can initiate a vertical scaling operation to just allocate, for example, 25 percent more memory than is currently allocated.

V. Discussion and Future Work

In the previous sections we saw the benefits that we expect from our proposal. Currently we are designing Infinity and there are some open questions and unexplored opportunities that we should address in order to fully realize our system.

How to manage scaling decisions: To allow applications to fully benefit from Infinity, we need to have clear mechanisms to guide selection of the most appropriate scaling primitives (III-B for a given scenario. For example, we need to decide if the allowed scaling primitives for a certain application should be decided and implemented by the programmer, or should another mechanism enable Infinity to automatically decide the ideal primitive when faced with a resource contention scenario? We aim to answer this question in a future work.

Meeting SLOs: Redirecting flows for processing by another entity may introduce latency to the NFs. We argue that different applications have different latency requirements, a premise that Infinity can leverage to select prioritized flows for processing by the local NF while only redirecting lower priority traffic to remote NF instances. This would ideally avoiding overloads on hardware components while maintaining SLOs.

Leveraging SmartNICs: In-network application SmartNICs can also benefit from the VIS abstraction. We plan as future work to extend Infinity and enable SmartNICs to participate on the VIS abstraction. This has the benefit of allowing further extension of resources for running high-demand in-network applications. For example, once a cryptography accelerator on a SmartNIC starts to suffer from queue build-up, Infinity can instruct the NF leveraging this feature to start sending a portion of its flows to an idle SmartNIC inside its VIS by leveraging the horizontal scaling abstraction. Another benefit of this extension will be to add SmartNICs processing capabilities to a hybrid VIS composed by both switches and SmartNICs, allowing for an optimal infrastructure utilization design.

VI. Conclusion

In this vision paper we described Infinity, a system that allows in-network applications to be deployed on top of a programmable switch fabric with virtually infinite resources. We see Infinity as an important step towards the next-generation dynamic network; an architecture to support new in-network applications while enabling increased performance, scalability and flexibility that current static solutions cannot provide.

References

[1] Production Quality, Multilayer Open Virtual Switch. https://www.openvswitch.org/.

[2] Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. http://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[3] 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 Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[4] 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. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[5] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, 2010.

[6] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014.

[7] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *NSDI*, pages 649–667, 2017.

[8] 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 Symposium on Operating Systems Principles (SOSP)*, 2017.

[9] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 109–114, 2012.

[10] E. Keller and J. Rexford. The 'Platform as a Service' model for networking. In *INM/WREN*, Apr. 2010.

[11] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 90–106, 2020.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[13] 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 the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[14] L. Peterson, S. Shenker, and J. Turner. Overcoming the internet impasse through virtualization. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, 2004.

[15] A. Sapio, M. Canini, C. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. *CoRR*, abs/1903.06701, 2019.

[16] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo. Flightplan: Dataplane disaggregation and placement for p4 programs.

[17] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang. Accelerated service chaining on a single switch asic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 141–149, 2019.

[18] L. Yu, J. Sonchack, and V. Liu. Mantis: Reactive programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 296–309, 2020.

[19] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proc. ACM SIGCOMM*, 2020.