Infinity: A Scalable Infrastructure for In-Network Applications

Marcelo Abranches
University of Colorado - Boulder
Boulder, USA
marcelo.abranches@colorado.edu

Karl Olson
University of Colorado - Boulder
Boulder, USA
karl.olson@colorado.edu

Eric Keller
University of Colorado - Boulder
Boulder, USA
eric.keller@colorado.edu

***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. 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. While the research community followed with innovative solutions, the ossification served as an impediment to adoption [13]. Today, this problem is even more pronounced with the rise of edge computing and applications which 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], [13] to programmable network hardware [11], [5], [4]. What this means for the next-generation Internet is that unique opportunities for in-network computing capabilities with high-performance and low-latency are now possible. In recent years, in-network applications have reduced latency for key-value stores [7], supported distributed locking [17], and performed data aggregation to accelerate machine learning [14]. These all point to the great potential for in-network computing that comes with new programmable hardware architectures.

Unfortunately, there's a catch. The underlying hardware which supports these efforts have fixed resources. 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 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 similar abstractions to meet the growing needs of in-network applications and dynamic architectures.

978-3-903176-32-4 © 2021 IFIP

Methods to address these resource limitations do exist [10], but incur significant costs for latency, which is fundamentally at odds with high speed network architectures. To address this challenge, we propose *Infinity*, a programmable network architecture which abstracts networking hardware into virtual aggregated hardware sets, giving in-network applications the illusion of having infinite processing capacity. Infinity ensures that in-network applications can locate resources within the aggregated hardware sets necessary to meet processing objectives by leveraging data plane dis-aggregation, scale-out techniques (vertical and horizontal), and per-function tailored performance requirements.

With Infinity we enable the following contributions to improve current in-network applications:

- We provide abstractions for building scalable and flexible in-network applications on top of programmable hardware, enabling high-performance and low-latency processing.
- We provide a method to dynamically assign and utilize system resources across a network-wide resource pool.

In the remainder of this paper, we detail the architectural solution afforded by Infinity before concluding with future directions to further enable a fully flexible internet architecture.

II. Programmable Network Hardware Resource Limitations and Disaggregation Challenges

Modern networks operate in a highly dynamic environment of competing stakeholder interests. Here, architects are pressured to provide solutions which support 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. The

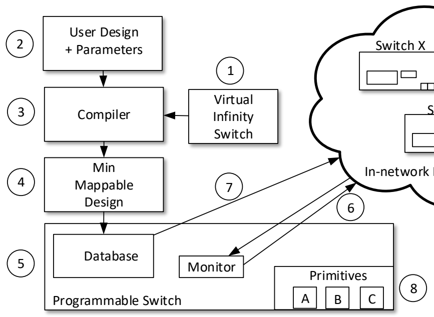

Fig. 1: **Infinity - High Level Deployment** (1,2) Infinity virtual switch architecture is integrated with user defined parameters via P4 Language. (3,4) Compiler prepares system resulting in minimal mappable architecture and loaded onto programmable switch. (5,6) Switch runs Infinity architecture, collecting in-network resource metrics from participating systems. (7) When action requires resources beyond capability of host, Infinity references resource pool and (8) employs one of its scaling primitives (subsection III-B) as needed.

versatility provided by programmable hardware therefore appeals to the modern network, enabling tailored functionality to meet performance, flexibility, energy efficiency, or cost effective business objectives. To bridge the gap between low-level hardware configuration and operator knowledge, domain specific languages (DSLs) like P4 [5] were introduced. Such languages enable administrators to programmatically define packet processing logic, enabling a flexible and feature complete data plane.

While programmable hardware offers significant customization with packet processing capabilities, resource limitations impose constraints on in-network processing scalability and desired functionalities. For example, modern programmable switches are limited in both fast memory resources and processing units (i.e., SRAM and ALUs) [12], [10], [15]. This resource limitation constrains in-network applications that rely on complex logic while simultaneously trying to maintain high-performance and scalability of network processing needs [16]. Methods to address some of these limitations exist, such as extending memory through a process based on remote direct memory access (RDMA), but incur both a buffering and round-trip processing cost [10]. Similar solutions to disaggregate processing resources do not yet exist, requiring a packet to return to the same entity for further processing.

III. Introducing Infinity

To enable a new level of performance, scalability and flexibility for in-network applications, we propose Infinity, a system which disaggregates multiple programmable networking hardware components into virtual aggregated hardware sets. Infinity gives applications the illusion of having infinite processing and memory capacity (limited in scale to an administrator defined hardware resource pool), enabling developers the ability to build in-network applications without consideration for system resource constraints.

Figure 1 shows the high level representation of Infinity's design. First, Infinity introduces a compiler that has a *Virtual Infinity Switch* (VIS) as its target. This compiler is responsible for generating the minimal mappable design for a given in-network application. Further, the VIS abstraction enables an in-network application to dynamically expand the resource boundaries of a single switch, ensuring that the developer need-not care about constraining the processing logic to fit within a single bounded entity.

With a viable, minimally mapped design, the operator can, at load time, provide that design to an Infinity controller, which will in turn place the application onto available resources. Infinity will then continue to monitor the deployed in-network applications to identify bottlenecks, such as SRAM or TCAM exhaustion. Once a bottleneck is identified, Infinity will employ one of its primitives (presented later in this section) to expand available resources for the in-network application.

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 [9] and Single Big Switch [6], [1] to introduce the VIS, an idealized abstraction of a programmable data plane with "infinite resources" (memory, processing, etc.).

After compilation, in-network applications are mapped to a pipeline of programmable forwarding engine stages (or other processing elements) that implement the desired logic. In a typical programmable switch case, these applications would map to a single device which is resource constrained and may limit either the capability or deployability of an application.

To enable a set of switches to appear as single entities with virtually infinity resources, each physical switch inside a VIS has a base functionality which supports an overlay network that encapsulates the original packets and transparently forwards them to a destination switch for further processing as needed. The overlays are built on top of a light-weight forwarding logic on each switch (i.e., custom header parsing, lookup ID, and forwarding). This enables flexible composition of primitives that support scaling hardware resources to meet application demands.

Each switch is also assumed to be partially reconfigurable. That is, the switch does not require the entire programmable hardware to be flashed in its entirely or to be taken offline during data plane reconfiguration. In this manner, Infinity can dynamically allocate processing resources on the switches without disrupting the currently deployed packet processing applications. 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 requires a collection of primitives to address a dynamically scaling design. These primitives enable flexible composition of hardware elements, allowing for dynamic scaling of resources.

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 dynamically increase processing stages, allowing for logic which expands the processing capabilities of a single switch (i.e., applications can use more physical pipeline stages than is available on a single switch).

To realize this, we require a mechanism to connect the segments of the pipeline spanning multiple switches. Figure 2 shows how Infinity realizes this primitive. Here, a given switch on the pipeline processes a packet and at the point the cut was made (to make more physical pipeline stages available), 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 is horizontal scaling. Similar to 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 (as one example) for a given application. Horizontal scaling is allowed at two levels: the pipeline, where the full processing pipeline is replicated on another switch, or pipeline segment level where a pipeline is partially replicated on another switch.

To realize this, we include additional logic to connect to the replicas, as shown in Figure 3. Here, a pipeline (or segment) is first replicated to another switch. Then, a load balancing element in the pipeline appends the preceding segment to the horizontally scaled segment. One complicating factor is that Infinity needs to know how the traffic was split - what traffic should go to each replica. For this, the application should specify a key upon which to partition memory resources, such as a 5-tuple representing the flow.

A second complicating factor is that the controller needs to know when resources are exhausted. For example, to detect memory exhaustion on a processing stage, we encode each memory entry with an extra bit to mark the resource as 'used' or 'unused'. For example, in a dynamic NAT, as new entries are added on stateful memory (e.g., registers), 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: The final primitive is vertical scaling. In contrast to horizontal scaling, which adds extra replicas, vertical scaling makes individual instances larger. As an analogy, consider a virtual machine with 1GB of memory allocated to it. With vertical scaling we would grow this memory to 2GB. Vertical scaling can be desirable in cases where horizontal scaling is not possible, such as when a full replication is not possible but resources remain partially unused.

There are two ways this can be realized. This first is through disaggregation [10]. A second mechanism is through migration. For example, if an application cannot afford the performance penalty with dissaggregation, but local resource constraints cannot fulfill the requirement, we must move to a different switch (with a larger allocation). This would then require migrating both the data and associated state, which can be done in a live manner [8].

C. Orchestrating Infinity

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

IV. Example Use Case

Layer 4 Load Balancer: A layer 4 load balancer's main purpose is to serve as an entry point for a scalable service,

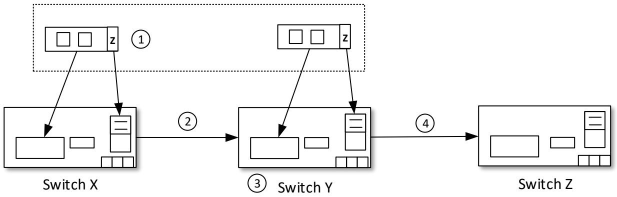

Fig. 2: **Infinity - Sequential Decomposition Operation** (1) The controller determines to sequentially decompose the pipeline and place the first segment on Switch X. (2) The controller informs Switch X that packets are to be encapsulated with the target header and forwarded according to the lookup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

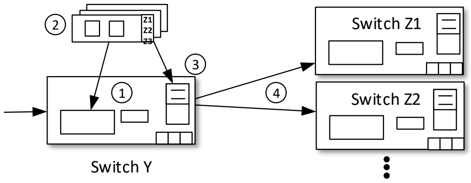

Fig. 3: **Infinity - Horizontal Scaling Operation** The Infinity controller determines a need to horizontally scale, placing replicas on switch Z1, Z2, etc. (1) Extra logic is added to Switch Y (the preceding segment), which will load balance across targets. (2) Controller sends the switch ID for Z1, Z2, etc. to Switch Y. This establishes encapsulation at Switch Y(3), which then forwards to replicas(4). When the packet arrives at a replica, it decapsulates the tag and processes the packet.

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 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 primitive to scale out.

V. Discussion and Future Work

While Ininity is a work in progress, we have identified potential gaps to fully realizing this vision. First, 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? Second, redirecting flows for processing by another entity may introduce latency to the network functions (NF). 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 avoid overloads on hardware components while maintaining SLOs. Third, in-network applications leveraging 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.

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. Further work and system details can be found within the author's websites [3].

Acknowledgments

This work was supported in part by NSF Grants 1652698 (CAREER) and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

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] M. Abranches et al. Infinity: A Scalable Infrastructure for In-Network Applications (extended version). https://github.com/mcabranches/Infinity/blob/main/paper/Infinity.pdf.

[4] P. Bosshart et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of ACM SIGCOMM*, 2013.

[5] P. Bosshart et al. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[6] M. Casado et al. Virtualizing the Network Forwarding Plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, 2010.

[7] X. Jin et al. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.

[8] E. Keller et al. Live Migration of an Entire Network (and its Hosts). In *Proceedings of ACM Workshop on Hot Topics in Networks*, 2012.

[9] E. Keller and J. Rexford. The 'Platform as a Service' Model for Networking. In *INM/WREN*, Apr. 2010.

[10] D. Kim et al. TEA: Enabling state-intensive network functions on programmable switches. In *Proceedings of ACM SIGCOMM*, 2020.

[11] N. McKeown et al. OpenFlow: Enabling Innovation In Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[12] R. Miao et al. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of ACM SIGCOMM*, pages 15–28, 2017.

[13] L. Peterson et al. Overcoming the Internet Impasse through Virtualization. In *Workshop on Hot Topics in Networks*, 2004.

[14] A. Sapio et al. Scaling Distributed Machine Learning with In-Network Aggregation. *CoRR*, abs/1903.06701, 2019.

[15] D. Wu et al. Accelerated service chaining on a single switch ASIC. In *Proceedings of ACM Workshop on Hot Topics in Networks*, 2019.

[16] L. Yu, J. Sonchack, and V. Liu. Mantis: Reactive programmable switches. In *Proceedings of ACM SIGCOMM*, pages 296–309, 2020.

[17] Z. Yu et al. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proc. ACM SIGCOMM*, 2020.