The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications

OLIVER MICHEL, Faculty of Computer Science, University of Vienna, Austria
ROBERTO BIFULCO, NEC Laboratories Europe, Germany
GÁBOR RÉTVÁRI, Budapest University of Technology and Economics (BME), Hungary
STEFAN SCHMID, Faculty of Computer Science, University of Vienna, Austria

Programmable data plane technologies enable the systematic reconfiguration of the low-level processing steps applied to network packets and are key drivers toward realizing the next generation of network services and applications. This survey presents recent trends and issues in the design and implementation of programmable network devices, focusing on prominent abstractions, architectures, algorithms, and applications proposed, debated, and realized over the past years. We elaborate on the trends that led to the emergence of this technology and highlight the most important pointers from the literature, casting different taxonomies for the field, and identifying avenues for future research.

CCS Concepts: • **Networks → Programmable networks**; **Programming interfaces**; *In-network processing*;

Additional Key Words and Phrases: Programmable data planes, network programmability, packet processing, in-network computation, programmable switches

ACM Reference format:
Oliver Michel, Roberto Bifulco, Gábor Rétvári, and Stefan Schmid. 2021. The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications. *ACM Comput. Surv.* 54, 4, Article 82 (April 2021), 36 pages.
https://doi.org/10.1145/3447868

1 INTRODUCTION

Computer networks are the glue of modern technological infrastructures. They are deployed in different environments, support a variety of use cases, and are subject to requirements ranging from best effort to guaranteed performance. This wide-spread use and heterogeneity complicate

The research leading to these results has received funding from the European Union.s H2020 Framework Programme (H2020-EU.2.1.1) under grant agreement no. 101017171 (Project "Marsal") and from the Austrian Science Fund (FWF) under project I 5025-N (DELTA), a joint project with the Hungarian National Research, Development and Innovation Office NKFIH, 2020-2023. Gábor Rétvári was funded by the NKFIH/OTKA Project # 135606. He is also with the MTA-BME Information Systems Research Group, the MTA-BME Network Softwarization Research Group, and Ericsson Research, Budapest.
Authors' addresses: O. Michel and S. Schmid, Faculty of Computer Science, University of Vienna, Währinger Str. 29, 1090 Vienna, Austria; emails: {oliver.michel, stefan_schmid}@univie.ac.at; R. Bifulco, NEC Laboratories Europe GmbH, Kurfürsten-Anlage 36, 69115 Heidelberg, Germany; email: bifulco@neclab.eu; G. Rétvári, Budapest University of Technology and Economics, Budapest, Magyar Tudósok Körútja 2, 1117 Hungary; email: retvari@tmit.bme.hu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0360-0300/2021/04-ART82 $15.00
https://doi.org/10.1145/3447868

the design of network systems and their main building blocks (i.e., network devices) in particular. While there is a pull towards specialization that allows network devices to be optimized for a particular task, there is also tension to make network devices commodity and general to reduce engineering cost. These opposing forces ultimately pushed the need for programmable networking equipment, allowing operators to change device functionality using a programming interface.

Programmability introduces a significant change in the relationship between device vendors and network operators. A programmable device frees the operator from waiting for the traditional networking equipment's years-long release cycles when rolling out new functionality. In fact, a new feature can be quickly implemented and rolled out directly by the operator using the device's programming interface. Moreover, programmability frees device vendors from designing equipment for a wide range of customer use cases; instead, they can invest engineering efforts into optimizing a set of well-defined building blocks that can be used to implement custom logic.

This new generation of programmable devices is proving especially helpful to operators who must now accommodate large-scale cloud computing, big data applications, massive machine learning, and the 5G mobile standard. These applications force operators to adopt new ways to architect communication networks, thus making **software-defined networking (SDN)**, edge computing, **network function virtualization (NFV)**, and service chaining the norm rather than the exception. Overall, this requires network devices such as switches, middleboxes, and **network interface cards (NICs)** to support continuously evolving and heterogeneous sets of protocols and functions on top of the already impressive set of features supported today, including tunneling, load balancing, complex filtering, and enforcing **Quality of Service (QoS)** constraints.

Supporting such an extensive feature set at the required flexibility, dynamicity, performance, and efficiency with traditional fixed function devices requires careful and expensive engineering efforts by device vendors. Such efforts involve the tedious and costly design, manufacturing, testing, and deployment of dedicated hardware components [124, 164], which introduce two main problems. First, rolling out new functionality incurs significant cost and is slow. This pushes vendors to support a given feature only when it becomes widely requested, impeding innovation. Second, implementing every single network protocol in a device's packet processing logic leads to inefficiencies, due to wasting valuable memory space, CPU cycles, or silicon "real estate" for features that only a small fraction of operators will ever use.

The introduction of programmable network devices addresses these issues, permitting the packet processing functionality implemented by a device to be comprehensively reconfigured. Interestingly, programmability is important both for software and hardware devices. On the one hand, new software-based network switches, running on general-purpose CPUs, provide reconfigurability through an extensive set of processing primitives out of which various pipelines can be built using standard programming techniques [76, 122, 131, 145, 157]. Leveraging advances in I/O frameworks [150, 156], these programmable software switches can achieve forwarding throughput on the order of tens of Gbit/s on a single commodity server. On the other hand, more challenging workloads, in the range of hundreds of Gbit/s, are in the realm of programmable hardware components and devices, like programmable NICs (SmartNICs) [35, 79, 195] and programmable switches [36, 38]. Similar to software switches, programmable networking hardware also offers various low-level primitives that can be systematically assembled into complex network functions using a domain-specific language [22] or some dialect of a general purpose language [49, 166].

While programmable data plane technologies have already gained substantial popularity and adoption, many questions around them remain unanswered. How do we adapt and use the elemental packet processing primitives to support the broadest possible selection of network applications at the highest possible performance? How do we expose the potentially very complex processing logic to the operator for easy, secure, and verifiable configuration? How do we abstract, replicate,

and monitor ephemeral packet processing state embedded deeply into this logic? What are the applications and use cases that benefit the most? Questions like these are currently among the most actively debated ones in the networking community.

Following the footsteps of Reference [94], this article provides a survey on the current technology, applications, trends, and open issues in programmable software and hardware network devices. We discuss available architectures and abstractions together with employed designs, applications, and algorithmic solutions. We imagine this article to be useful for a broad audience: researchers aiming at getting an overview of the field, students learning about this novel, exciting technology, and practitioners interested in academic foundations or emerging applications in programmable data planes. Finally, we provide an online reading list that will be continuously updated beyond the writing of this article [127]. Our focus is on the data plane and, in particular, on the reconfigurable packet processing functionality inside the data plane responsible for enforcing forwarding decisions; for comprehensive surveys on control plane designs and SDN, see References [14, 52, 101, 137, 190, 196].

The remainder of the article is organized as follows: In Section 2, we introduce the most important aspects of programmable data planes. Then, we elaborate on architectures and platforms in Section 3, before discussing abstractions and algorithms commonly leveraged in Sections 4 and 5. In Section 6, we present applications and proposed systems built on top of this technology. Finally, we highlight some of the most compelling open problems in the field in Section 7 before briefly summarizing the work discussed in this article through a taxonomy and conclusion in Section 8.

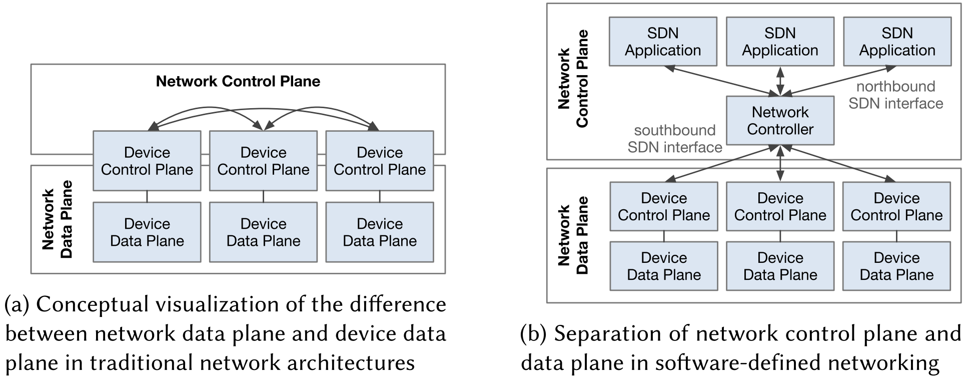
2 THE PROGRAMMABLE DATA PLANE

Before diving deeper into this survey, we will now give a brief overview of the various developments that led to the need for data plane programmability. We will also describe what the responsibilities of the data plane are and what data plane programmability exactly means.

2.1 Control Plane – Data Plane Separation

Conventional network equipment, regardless of the implementation (e.g., pure software or specialized hardware) and function (e.g., a switch or an edge router), has its functionality logically split into a *device control plane* and a *device data plane*. The device control plane is in charge of establishing packet processing policies, such as where to forward a packet or how to rewrite its headers, and managing the device, including checking its health and performing maintenance operations. In turn, the device data plane is responsible solely for executing the packet processing policy set by the device control plane, usually with very high performance requirements. The control planes of the individual devices within a given network scope, such as an organizational domain or the entire Internet, interact through a distributed routing protocol. As depicted in Figure 1(a), through this interaction they create the illusion of a single *network-level control plane*, executing a virtual global packet forwarding policy in a distributed fashion.

With the introduction of the **Software-defined Networking (SDN)** paradigm [52, 196], the network control plane has emerged as a separate entity, a logically centralized *controller*, with some of the device control plane functions separated out and moved to this network-level functionality. The network control plane is in charge of *(i)* maintaining an inventory of the devices in the data plane; *(ii)* accepting high-level, network-wide policies (or *intents*) through a northbound controller interface; *(iii)* compiling these high-level intents to per-device packet processing policies; and finally, *(iv)* programming these policies into the individual devices through a southbound controller interface. In this architecture, the individual switches do not need to implement the logic required to maintain packet forwarding policies locally (e.g., they do not run routing protocols to build routing tables); rather, they get these policies prefabricated from the network control



(a) Conceptual visualization of the difference between network data plane and device data plane in traditional network architectures

(b) Separation of network control plane and data plane in software-defined networking

Fig. 1. Traditional vs. SDN-based network architectures.

plane. Here, the controller-switch communication occurs through a standardized southbound API and protocol, like OpenFlow [125], ForCES [189], P4Runtime [34], or the Open vSwitch Database Management Protocol [143]. This architecture is depicted in Figure 1(b). Note, however, that the device control plane does not fully disappear in the SDN framework; rather, it remains in charge of terminating control channel towards the remote network control plane and managing the device data plane.

2.2 Data Plane Functions

A device's data plane processes network packets by performing a series of operations, including the parsing of (a subset of) the packet, determining the sequence of processing operations that need to be applied, and forwarding it based on the results of such operations. Packet processing entails the following basic functional steps: *parsing*, *classification*, *modification*, *deparsing*, and *forwarding*. On top of the basic functionality, most packet processing systems can provide additional services, such as *scheduling*, *filtering*, *metering*, or *traffic shaping*.

Parsing is the process of locating protocol headers in the packet buffer and extracting the relevant header fields into packet descriptors (metadata). These values are then used during *classification* to match the packet with the corresponding forwarding policy, which describes the forwarding or processing actions to be applied to the packet (e.g., which output port to use or whether to drop the packet) and the required packet modification actions (e.g., rewriting a header field). The *modification* step applies the actions retrieved during classification, and may also include the update of some internal state (e.g., to increase a flow counter). Once all modifications are applied, packets are re-generated from packet descriptors (*deparsing*), and finally in the *forwarding* step the packet is sent to a port for transmission. This step may include the application of *scheduling policies* (e.g., to enforce network-level QoS policies), and *traffic shaping* to limit the amount of network resources a flow/user may consume. The combination of classification and subsequent processing based on matched rules is commonly referred to as *match-action processing* (see more in Section 4.1.2).

Generally, these steps happen in the reported order. Depending on the implementation and desired functionality, however, processing steps may be repeated by sequencing multiple match-action cycles after each other or by recirculating a packet to the beginning of the pipeline.

2.3 Data Plane Programmability

With the emergence and adoption of the SDN paradigm, device functionality has become much more flexible and dynamic. As previously explained, in conventional network equipment the data

plane functionality is deeply ingrained into the device hardware and software, and hence generally cannot be changed during the lifetime of the device. For software-based packet processing systems, major vendor software updates are required to change data plane functionality. This fixed functionality affects virtually all data plane operations. The format and semantics of the entries that can be loaded into match-action tables are fixed; devices only understand a finite set of protocol headers and fields. For example, an Ethernet switch does not process layer-3 fields and an antiquated router will not support IPv6 or QuiC. The types of processing actions that can be applied and the order in which these are enforced are set by the device vendor; typically, MAC processing is followed by an IP lookup phase before enforcing ACLs and performing group processing. For example, this makes it impossible to apply IP routing lookup on packets decapsulated from VXLAN tunnels. Finally, queuing disciplines (e.g., FIFO or priority queuing only, without support for, e.g., BBR [28]) and the type of monitoring information available from the data plane are predetermined.

Through SDN and the emergence of increasingly more general hardware designs, today's data plane devices can be reconfigured from the network control plane, either partially or in full. This development has motivated the introduction of the term *programmable data plane*, referring to the new breadth of network devices that allow the basic packet processing functionality to be dynamically and programmatically changed. In the context of this survey, we use the following definition for the programmable data plane:

> Data plane programmability refers to the capability of a network device to expose the low-level packet processing logic to the control plane through a standardized API, to be systematically, rapidly, and comprehensively reconfigured.

It is important to note that data plane programmability is not a binary property. Up to some degree, configuring a conventional "fixed-function" device can be viewed as data plane programming. As the exact boundaries between data plane configuration and programmability are still actively debated in the community [8, 123], in the following discussion, we embrace an inclusive interpretation of the term and lay the emphasis on the comprehensiveness of the types of modifications a device allows on the packet processing functionality. Correspondingly, we focus on the following aspects:

- *new data plane architectures, abstractions, and algorithms* that permit the data plane functionality to be fully and comprehensively reconfigured, including the parsing of new packet header fields, matching on dynamically defined header fields, and exposing new packet processing primitives to the control plane, which together facilitate the deployment of even completely new network protocols; and
- *new applications that can be realized in the data plane through programmability*, including monitoring and telemetry, massive-scale data processing and machine learning, or even complete key-value stores implemented fully inside the network devices with zero or minimal intervention from the control plane.

3 ARCHITECTURES

While data plane programmability was initially mostly targeted at switches (especially in data centers), today a wider range of devices and functions allow for low-level programmability. Programmable data plane hardware or software is not only used for packet switching, but also increasingly for general network processing and middleboxes (e.g., firewalls or load balancers) [45, 110, 119]. Additionally, programmable NICs enable data plane programmability at the edge of the network. These devices can be realized on top of one or multiple of the several different architectures.

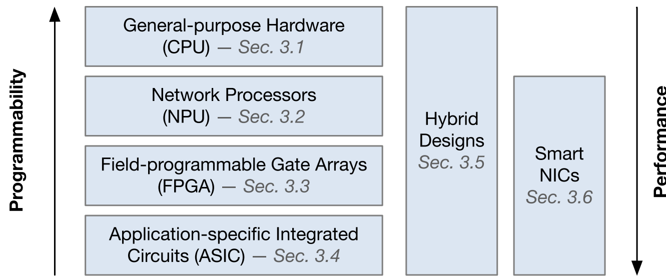


Fig. 2. Overview of hardware architectures programmable data plane systems are commonly built upon.

In hardware designs, data plane functionality may be implemented in an **Application-specific Integrated Circuit (ASIC)** [36, 38], a **Field-programmable Gate Array (FPGA)** [54, 195], or a network processor [35, 79]. These platforms generally offer high performance due to dedicated and specialized hardware components, such as **Ternary Content Addressable Memory chips (TCAM)** for efficient packet matching. A software data plane device, however, is one where the data plane executes the entire processing logic on a commodity CPU [71, 140, 145, 165] using optimized algorithms and data structures [53, 98, 176]. Yet, the distinction between hardware and software data planes is somewhat blurred. For instance, a hardware-based device may still invoke a general-purpose CPU to run functions that are not supported natively in the underlying hardware or do not require high performance. Similarly, modern software switches rely on the assistance of domain-specific hardware capabilities for efficiency reasons, such as **Data Direct I/O (DDIO), Receive Side Scaling (RSS)**, and increasingly SmartNIC offloads to run packet processing logic partially or entirely in hardware. Next, we present an overview of the main design points in architectures for programmable data plane systems together with their characteristics, use cases, and tradeoffs made. The high-level relationship between the different sections is depicted in Figure 2.

3.1 General-purpose Hardware

General-purpose hardware architectures and CPUs (like x86 or ARM) commonly used in commodity servers and deployed in data centers at massive scales support a wide range of packet processing tasks. For example, efforts of telecom operators towards advancing the 5G cellular network standards and network function virtualization [119, 138] rely on the capability to perform high-performance packet processing with general-purpose servers [95, 140]. Modern virtualized data centers usually have servers running the network access layer [100, 142], using a software switch that connects virtual machines to the physical network [12, 131, 145, 179]. Driven by these requirements, over the past years software-based packet processing has made significant inroads in the traditionally hardware-dominated network appliance market [50, 68, 147] with several established programmable software switch platforms for efficient network virtualization (Open vSwitch [145], VPP [12], BESS [71], NetBricks [140], PacketShader [72], and ESwitch [131]), user space I/O libraries (NetMap [156], DPDK [150], FD.io [151], and XDP with eBPF [74]) and NFV platforms [91, 102, 178, 182].

At a high level, packet processing in a server is a simple process that includes copying the packet's data from a NIC buffer to the CPU, processing it for parsing and modification/update steps before copying or moving the data again to another NIC buffer or to some virtual interface [110]. In practice, this process is significantly more cumbersome due to the complex architecture of modern server hardware, whereby achieving high performance for networked applications requires accounting for the architecture and characteristics of the underlying hardware [4].

To accelerate network packet input and output, several shortcuts in the path a packet takes from the wire to the CPU both in software and at the hardware-level exist. In software, kernel-bypass networking can be used to map the memory area used by NICs to write packets to or read packets from directly into user space. This eliminates costly context switches and packet copies vastly improving networking performance compared to standard sockets. Applications using kernel-bypass frameworks, such as NetMap [156] or DPDK [150], however, cannot use any kernel networking interfaces and must implement all packet processing functionality they may need (e.g., a TCP stack or routing tables). The **Express Data Path** in the Linux kernel **(XDP)** [74] alleviates this problem by allowing packet processing applications to be implemented in a constrained execution environment in the kernel while using some of the OS host networking stack. At the hardware-level, modern NICs implement *Data Direct I/O* [37, 51] to copy a received packet descriptor directly into the CPU L3 cache, bypassing the comparatively slow main memory.

Given the above hardware properties and constraints, software implementations apply a number of techniques to efficiently use the available resources [4, 110, 145]. Packets are usually processed in batches to amortize the cost of locks on contended resources across the processing pipeline and to improve data locality [12]. Other typical techniques include adopting data structures that minimize memory usage to better fit in caches [155], aligning data to cache lines to avoid loading multiple cache lines for few additional bytes [131], and distributing packets across different processors keeping flow affinity to avoid cache synchronization issues [91, 178].

Apart from these general optimization techniques, a software implementation can use several further optimization strategies to accelerate packet processing [110]. For instance, ClickOS [122], FastClick [13], and BESS [71] implement a run-to-completion model, in which each packet is entirely processed before processing a second packet on the same core, whereas NFVnice [102] uses standard Linux kernel schedulers and backpressure to control the execution of packet processing functions. In contrast, VPP [12] leverages pipelined processing, performing each single processing step on the entire batch of packets, before starting the next processing step. Likewise, parsing, classification, and modification/update steps can be intertwined as needed and desired by the programmer [13, 71]. Lazy parsing can be employed to avoid unnecessary and costly parsing operations (e.g., for packets that are to be dropped early). All these different approaches are possible due to the flexibility of general-purpose CPUs, which do not mandate any specific processing model.

With the emergence of specialized accelerators for offloading packet processing and the resulting hybrid designs, we might see fewer pure software implementations of network functions, especially for switching and virtualization use cases. Yet, we believe that efficient software-based packet I/O and processing will remain crucial for almost all network and cloud applications, and even become more important for applications such as high-performance web servers, container frameworks, or analytics engines. Finally, the flexibility and cost benefits of NFV approaches highlight the continued importance of software packet processing.

3.2 Network Processors

Network processors, sometimes referred to as **Network Processing Units (NPUs)**, are specialized accelerators, usually employed both in switches and NICs. Unlike general-purpose hardware, NPU architectures are specifically targeting network packet processing. Devices usually contain several different functional hardware blocks. Some of these blocks are dedicated to network-specific operations, such as packet load balancing, encryption, or table lookups. Some other hardware resources are instead dedicated to programmable components that are generally used to implement new network protocols and/or packet operations.

Given its availability for research and the support for recent data plane programming abstractions, we will describe the architecture of a **Netronome Network Function Processor (NFP)**

programmable NIC as an example of a NPU [79]. Since network traffic is a mainly parallel workload, with packets belonging to independent network flows, network processors are generally optimized to perform parallel computations, with several processing cores. While the number of these cores could be in the order of tens or hundreds, the per-core computing power is usually limited, thus most of the performance benefits come from the ability to process many packets in parallel. The NFP architecture contains 72 such cores (so called micro engines) that each allow for 8 concurrent threads. Micro engines are directly co-located with fast SRAM banks of a few hundred KBs used to host frequently accessed data required for the processing of each network packet. Additionally, larger memories that host the forwarding tables and access control lists used by the networking subsystem to decide how to forward (or drop) a network packet are shared by all micro engines. All building blocks are interconnected via a high-speed switching fabric for low-latency communication between cores. Packets enter and exit the system through arrays of specialized cores for packet parsing, classification, and load balancing to the processing cores. The architecture supports different interface capacities up to 2×40 Gbit/s Ethernet. A PCIe interface enables communication to the system's CPU via **direct memory access (DMA)**.

Similar to general-purpose servers, network processors support a flexible programming model and do not mandate any particular order for the processing steps of a packet. Additionally, the entire packet is generally available for processing as data can be stored at the different levels of the processor's memory hierarchy enabling advanced applications operating on packet payloads, including, for example, deep packet inspection for intrusion detection.

3.3 Field-programmable Gate Arrays

FPGAs are semiconductor devices based on a matrix of interconnected, configurable logic blocks. Contrary to ASICs, FPGAs can be programmed and reconfigured after manufacturing to implement custom logic and tasks. While custom ASIC designs generally offer the best performance, modern FPGAs narrow this gap for many use cases due to increased clock speeds and memory bandwidth [106]. High-level synthesis or specialized compilers allow programming FPGAs using languages such as C or P4 as opposed to more complex and cumbersome hardware description languages, such as Verilog [80, 186]. The balance of high performance together with programmability make FPGAs not only interesting for prototyping, but also a powerful alternative to costly and rigid ASIC designs for production environments [24, 105, 136]. In the context of networking, FPGAs are primarily used on NICs to offload packet processing from servers with the goal of saving precious CPU cycles [54].

The availability and comparatively low cost compared to programmable ASICs make FPGAs particularly interesting for academia to prototype high-performance network data planes. NetFPGA, for example, is a widely available open-source FPGA-accelerated NIC. The most recent version (FPGA SUME) couples a Xilinx Virtex 7 FPGA with four 10 Gb Ethernet ports [195]. A more recent effort in this direction is Corundum [55], which provides an open source platform for implementing a 100 Gbps NIC on FPGA. Corundum is a collection of the basic NIC modules and building blocks that are ready to be implemented on several commercial FPGA cards.

3.4 Application-specific Integrated Circuits

While in the early days of the ARPANET and the Internet, routing and packet processing were performed in software [73], the rapid adoption and increasing scale of the Internet required more efficient hardware-based designs to keep up with increasing packet rates. An ASIC is a chip specialized and optimized for (in this case) high-performance packet processing, focusing on implementing just the minimal set of operations required for this task. In fact, network devices built using ASICs generally include a second general-purpose sub-system (e.g., based on CPUs) to implement

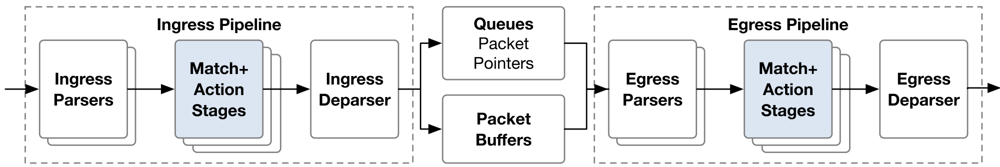


Fig. 3. The architecture of an RMT-like switching ASIC.

the device's monitoring and control functions, as well as more complex and uncommon packet processing functions that the ASIC does not support. Processing in ASICs is usually called the *fast path* and, by contrast, the *slow path* is the processing done by the general-purpose sub-system.

A typical ASIC is implemented as a fixed pipeline of different processing steps that are performed sequentially (e.g., L2 processing before L3 processing or MPLS lookup). Fast SRAM or TCAM banks alongside the pipeline store forwarding rules (such as routing entries) accessed in the individual lookup stages. Most high-performance switches and routers such as the Cisco ASR or Juniper MX series devices still leverage fixed-function ASICs. While extremely efficient, these devices suffer from long and costly development cycles, thus hindering flexibility and innovation.

As a result, more flexible and programmable switching chip architectures, such as **Reconfigurable Match-action Tables (RMT)** [23], **the Protocol-independent Switch Architecture (PISA)** [30], and implementations, such as Intel Flexpipe [36] or Intel Tofino [38] have been proposed. Programmable data plane devices allow network operators to programmatically change the low-level data plane functionality to support novel or custom protocols, implement custom forwarding or scheduling logic, or enable new applications entirely in hardware.

These RISC-inspired programmable ASICs are organized as a pipeline of programmable match-action stages. Before a packet enters the pipeline, a programmable parser dissects the packet buffer into individual protocol headers. The match-action stages then consist of memory banks implementing tables for matching extracted packet headers and **Arithmetic Logical Units (ALUs)** for actions such as modifying packet headers, performing simple calculations, or updating internal state. Furthermore, the tables may have different matching capabilities depending on the way they are implemented in hardware. For instance, exact matching tables can be implemented as hash tables in SRAM, while wildcard matching tables are generally implemented using more expensive TCAM. At the end of the pipeline, a deparser again serializes the individual (possibly altered) headers before sending the packet out on an interface or passing it to a subsequent pipeline. In many switches it is common to have at least two such pipelines: an ingress and an egress pipeline [30]. Figure 3 depicts the RMT reference design for programmable switches.

3.5 Hybrid Architectures

In addition to the platforms discussed above, new and interesting hybrid hardware-software designs mix existing concepts with fresh ideas from distributed systems and multi-processor design. While it is often believed that the performance of programmable network processors is lower than integrated circuits, there exists literature questioning this assumption and exploring these overheads empirically. In particular, Pongrácz et al. [147] show that the overhead of programmability can be relatively low. In benchmarks, the authors find throughput of NPUs either similar or only 30%–35% lower at comparable power consumption compared to their non-programmable NIC counterparts. Furthermore, the performance gap between programmable and hard-wired chips is not primarily due to programmability itself but rather because programmable network processors are commonly tuned for more complex use cases.

Past work on hybrid architectures also explores the opportunity to use **Graphics Processing Unit (GPU)** acceleration. For many applications, such as network address translation or analytics, packet processing workloads can be partitioned using a packet's flow key (e.g., IP 5-tuple). This makes packet processing a massively parallelizable workload, which could be in principle suitable to be implemented in multi-threaded hardware like GPUs [72]. However, the advantages and disadvantages of this strategy are actively debated in the systems community [63, 90]. Kalia et al. [90] argue that for many applications the benefits arise less from the GPU hardware itself than from the expression of the problem in a language such as CUDA or OpenCL that facilitates memory latency hiding and vectorization through massive concurrency. The authors demonstrate that when applying a similar style of optimizations to different algorithm implementations, a CPU-only implementation is more resource-efficient than the version running on the GPU. An answer to the issues raised by Kalia et al. was given by Go et al. [63]. Their work finds that with eight popular algorithms widely used in network applications, *(i)* there are many compute-bound algorithms that do benefit from the parallel computation capacity of GPUs, and *(ii)* the main performance disadvantage of GPUs comes from the need to traverse the PCIe bus to move data from the main memory to the GPU. Nonetheless, it should be noted that in Reference [63] there are several use cases that require some encryption algorithm to be run on the packet data. Today, these workloads are better handled with dedicated hardware provided both by CPUs and NICs, thereby reducing the potential areas of applicability of GPU-based acceleration for packet processing.

Various applications are particularly suitable for hybrid hardware-software co-designs. One of them is in the context of forwarding table optimization. In References [17, 92], architectures are studied that allow high-speed forwarding, even with large rule tables and fast updates, by combining the best of hardware and software processing. Specifically, the CacheFlow system [92] caches the most popular rules in a small TCAM and relies on software to handle the small amount of cache-miss traffic. The authors observe that one cannot blindly apply existing cache-replacement algorithms because of the dependencies between rules with overlapping patterns. Rather long dependency chains must be broken to cache smaller groups of rules while preserving the semantics of the policy.

Another example for applications that commonly leverage hybrid hardware-software designs are network telemetry and analytics systems. These systems must make difficult tradeoffs between performance and flexibility. While it is possible to run some basic analytics queries (e.g., using sketches) entirely in the data plane at high packet rates, systems generally follow a hybrid approach where analytics tasks are partitioned between hardware and software to benefit from high performance in hardware, as well as from programmability, concurrent measurement capabilities, and runtime-configurable queries in software. Systems employing such a design are *Flow [175], Sonata [69], and Marple [135]. We further elaborate on these systems in Section 6.1.

In conclusion, we witness a trend towards more specialization and, as a result, more hybrid architectures. We elaborated on two areas where researchers have proposed hybrid designs in the past; given the vast spectrum of flexibility and performance across the different platforms, we believe there will be more hybrid approaches across almost all network systems in the future.

3.6 Programmable NICs

Orthogonal to the previously presented architectures, programmable NICs, a new platform for programmable data planes, have attracted significant attention in the networking community. These devices (often referred to as SmartNICs) are commonly built around NPUs and FPGAs. The design and operation of programmable NICs involve a range of interesting aspects related to the host-network communication interface and operating system integration they provide. SmartNICs are

consequently well-suited for offloading end-to-end mechanisms (e.g., congestion control) and applications, such as key-value stores and virtualization.

Modern non-programmable NICs already implement various comparatively advanced features in hardware, such as protocol offloading, multicore support, traffic control, and self-virtualization. Programmable NICs go a step further by enabling custom packet processing and are programmable in subsets of general-purpose languages [79] or specialized data plane programming abstractions, such as P4 [22] or eBPF. In the following, we only focus on the architectural aspects of such SmartNICs and defer applications leveraging these devices to Section 6.

Despite its promising characteristics, SmartNICs are still trailing in adoption due to various challenges related to the development process of applications as well as ensuring efficiency of those applications on this novel platform. The development abstractions are, in particular, a concern for server applications that offload computation and data to a NIC accelerator. Floem [146] is a set of programming abstractions for NIC-accelerated applications that simplify data placement and caching, partitioning of code for parallelism, and communication strategies between program components across devices. It also provides abstractions for logical and physical queues, global per-packet state, remote caching, and interfacing with external application code.

Related to the development challenges, it remains unclear (especially in distributed applications) how functionality should be offloaded to maximize overall efficiency and benefits. Toward answering this question, Liu et al. propose iPipe [113], a generic actor-based offloading framework to run distributed applications on commodity SmartNICs. iPipe is built around a hybrid scheduler that combines different scheduling policies to maximize device utilization.

An interesting distributed application and use case for SmartNICs is to run microservices on SmartNIC-accelerated servers. By offloading suitable microservices to the SmartNIC's low-power processors, one can improve server energy-efficiency without latency loss. A system leveraging this approach is E3 [115], which follows the design philosophies of the Azure Service Fabric microservice platform and extends key system components to a SmartNIC. E3 addresses challenges associated with this architecture related to load balancing workloads, placing microservices on heterogeneous hardware, and managing contention on shared SmartNIC resources.

Going forward, we believe SmartNICs will have a great impact across a wide range of traditionally software-based applications and mechanisms, such as programmable congestion control, TCP/TLS connection termination, or network virtualization. Offload to SmartNICs can introduce significant cost savings in such scenarios by freeing up precious CPU cycles. We expect to see more host-based services, such as firewalls, L7 gateways, or hypervisor-based load balancing being offloaded to SmartNICs. Efficient host-based data plane programming and the capability to offload applications transparently to SmartNICs will further accelerate this trend [26]. *Takeaways* — In this section we introduced the wide range of platforms and architectures upon which programmable data plane systems are built. This range spans from highly programmable but comparatively slow general purpose CPUs to ASICs that expose a rigid programming model with constrained resources but offer unparalleled performance. As alluded to in Section 3.5, we expect to see more hybrid architectures. Depending on the constraints imposed by the workload, carefully partitioning a system between various architectures has the potential to provide the best of several worlds. Developing best practices and being able to partition workloads systematically or even dynamically will be critical going forward. Furthermore, we expect to see this broad spectrum of platforms to be also available in public cloud infrastructures. While all previously discussed architectures are available and deployed by operators, low-level data plane programmability has not yet been widely virtualized and exposed to cloud customers. We do, however, see trends in this direction with, for example, **Amazon Web Services (AWS)** allowing for kernel-bypass technologies on

instances with their *enhanced networking* offering. AWS also offers VMs equipped with FPGAs that are directly connected to the network.

4 ABSTRACTIONS

The differences among data plane technologies are often reflected in the packet processing primitives exposed to the control plane and language constructs that can be used to combine these primitives to implement the required pipeline. Given this inherent architectural coupling, we next discuss common abstractions used and exposed in programmable data plane systems. We start by discussing programmable packet processing pipelines before diving deeper into abstractions for packet parsing and scheduling. Finally, we review programming languages and compilers.

4.1 Programmable Packet Processing Pipelines

Flexible packet processing is the core capability of programmable data planes. Today's packet processing pipelines are generally built on top of two fundamental abstractions: the data flow graph abstraction and the match-action pipeline abstraction.

4.1.1 Data Flow Graphs. Early designs for packet processing systems borrowed largely from generic systems design [177] and machine learning [1], adopting the data flow graph abstraction to architect programmable switches [99]. This model is also heavily used in stream processing frameworks such as Apache Flink or Spark. A data flow graph describes processing logic as a graph, with nodes representing elemental computation stages and edges representing the way data moves from one computation stage to another. A favorable property of this abstraction is its simplicity, allowing the programmer to assemble a well-defined set of processing nodes into meaningful programs using a familiar graph-oriented mental model. This way, computational primitives (nodes) are developed only once and can then be freely reused as many times as needed to generate new modular functionality creating a rapid development platform.

Perhaps the earliest programmable switch framework adopting the data flow graph abstraction was the Click modular software router [99]. The unit of data moving through the Click graph is a network packet on which nodes can perform simple packet processing operations, such as header parsing, checksum computation and verification, field rewriting, or checking against ACLs. Some nodes provide network protocol-specific functions, such as handling ARP requests and responses, while others offer more general data flow control functions, such as load balancing, queueing, or branching (selecting the next processing stage out of several alternatives).

ClickOS [122], FastClick [13], **Vector Packet Processing (VPP)** from the FD.io project [151], the **Berkeley Extensible Software Switch (BESS)** [71], and NetBricks [140] adopt a similar design, with the difference that the fundamental data unit moving along the data flow graph is now a vector of packets instead of a single packet. This development stems from the observation that batch-processing amortizes I/O costs over multiple packets and that using built-in vector instruction sets of modern CPUs results in more efficient software implementations [13, 72, 150]. In addition, NetBricks introduces a new framework for the isolation of potentially untrusted packet processing nodes, using novel language-level constructs and zero-cost compile-time abstractions [140].

The presence of user-defined functionality abstracted as data flow graph nodes gives great flexibility and extendibility [104, 122]. At the same time, this flexibility tends to make the resulting designs piecemeal, and heterogeneity complicates high-level network-wide abstractions and encumbers performance optimization [107, 108].

4.1.2 Match-action Processing. The match-action abstraction describes data plane programs using a sequence of lookup tables (flow tables) organized into a hierarchical structure [22, 125, 131,

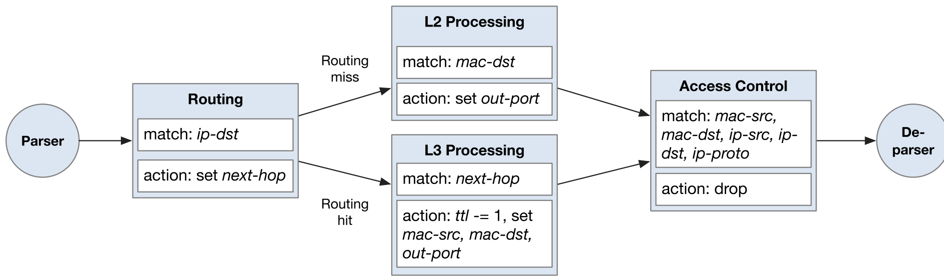


Fig. 4. Simplified match-action table dependency graph for a basic router (inspired by Fig. 3 in Reference [22]).

145, 165]. A subset of the packet header fields is used to perform a table lookup to identify the corresponding packet processing actions, which can then instruct the switch to rewrite packet contents, encapsulate/decapsulate tunnel headers, drop or forward the packet, or defer packet processing to subsequent flow tables. The programmer configures the packet processing behavior through dynamically setting the content of the flow tables, by adding, removing, or modifying individual entries with the associated matching rules and processing actions via a standardized API [144]. This has the benefit of exposing reconfigurable data plane functionality to operators using the familiar notion of *flows* described by matching *rules* defined over header fields, an abstraction extensively used in firewalls and ACLs. Hierarchies of lookup tables, as also used by conventional fixed-function router ASICs, are used to synthesize more complex L2/L3/L4 pipelines.

The match-action abstraction was popularized for programming switches by the OpenFlow protocol [125], which borrowed greatly from Ethane [29]. OpenFlow in its first version allowed the definition of only a single flow table using a rather limited set of header fields; the abstraction was later extended to a pipeline of multiple flow tables defined over a large array of predefined header fields. With the introduction of multi-table match-action pipelines in the OpenFlow v1.1 specification, the distinction between the data flow graph and the match-action abstractions has become increasingly blurry [125]. As illustrated using an example in Figure 4, a hierarchical match-action pipeline can easily be conceptualized as a special data flow graph with lookup tables as processing nodes and "goto-table" instructions as the edges.

Currently Open vSwitch [145] remains the most popular OpenFlow software switch, using a universal flow-caching based datapath for implementing the match-action pipeline. This design was improved upon by ESwitch [131], which introduces data plane specialization and on-the-fly template-based datapath compilation to achieve line-rate OpenFlow software switching. Despite being widely adopted, OpenFlow is limited in matching arbitrary header fields. This sparked research in flexible lookup tables with rich semantics, configurable control flow, and platform-specific extensions.

Driven by the advances in switching ASIC technology, the **Reconfigurable Match Tables (RMT)** abstraction [23] overcomes the main limitations in OpenFlow ASICs in two ways: by letting match-action tables to be defined on arbitrary header fields, and extending the previously rather limited set of packet processing actions available. While RMT allows for matching on arbitrary bit ranges within a packet header and applying modifications to the packet headers in a programmable manner, applications for this architecture are still constrained by the rigid sequential design of the architecture. dRMT [32] relaxes some of these sequential processing constraints and provides a more flexible architecture by separating memory banks for matching packets from processing stages. This design allows using hardware resources more efficiently and, compared to RMT, increases the set of programs mappable to line-rate hardware architectures. Lately, P4 [22]

and the accompanying hardware and software switch projects [36, 38, 165] have been met with increasing enthusiasm from the side of device vendors, operators, and service providers [57].

4.2 Stateful Packet Processing

In the early days of the Internet, most stateful packet processing has taken place at the end hosts (e.g., to terminate a TCP connection), while most packet forwarding and processing within the network operated in a stateless manner (i.e., devices do not need to keep track of any state between packets). Today, stateful network functions are commonplace and include firewalls, network address translators, intrusion detection systems, load balancers, and network monitoring appliances [184]. With the emergence of high-performance packet processing capabilities in software, network functions are routinely implemented in commodity servers, an approach referred to as **network function virtualization (NFV)**. More recently, programmable line rate switches allow for comprehensive programmability. As a result, these devices are commonly used for tasks other than switching and routing. We will discuss examples of new use cases and applications in Section 6.

4.2.1 Programming Abstractions for Stateful Packet Processing. Providing flexible and platform-independent programming abstractions for stateful packet processing on programmable data plane devices remains a major challenge today. Due to the complexities and constraints associated with most platforms, stateful packet processing is often still implemented in SDN controllers, significantly reducing overall network performance. Toward this problem, several works propose abstractions around **finite state machines (FSM)** for simplified programming of stateful packet processing pipelines. Data plane programs defined using the FSM abstraction can then be compiled for and offloaded to line rate hardware devices [15, 16, 134, 148]. Other more language-focused approaches include Domino [172], which introduces the abstraction of *packet transactions* that allows expressing stateful data plane algorithms in a C-like language without having to define match-action tables or other architecture-related details. Hardware designers can specify their instruction sets through small processing units called atoms that the Domino compiler configures based on the application code. The work on Domino also provides a machine model for programmable line-rate switches, called Banzai machine, that can be used as a target for Domino programs and is available to the community. While Domino programs target a single switch, SNAP [9] allows programmers to develop stateful networking programs on top of a "single switch" network-wide abstraction. The SNAP compiler handles how to distribute, place, and optimize access to state arrays across multiple hardware targets. Finally, SwingState [118] is a state management framework that enables consistent state migration among programmable data planes by piggybacking state updates to regular network packets. A static analyzer for the P4 language detects which state needs to be migrated and augments the code for in-band state transfer accordingly.

While FSMs provide a naturally suited abstraction for stateful packet processing, realizing scalable stateful packet processing systems based on programmable data plane systems is still challenging and appears to be one factor hindering the adoption of programmable data plane technology. In particular, realizing low-latency stateful applications in programmable ASICs is cumbersome due to target-specific requirements and constrained memory and stateful ALU resources.

4.2.2 State Management in Virtualized Network Functions. NFV promises simplifying middlebox deployment, improving elasticity and fault tolerance while reducing costs [102, 139]. In practice, however, it remains challenging to deliver on these promises due to the tight coupling of state and processing in NFV environments. State either needs to be shared among NF instances or is kept local for a certain subset of network flows. In either way, keeping network-wide state consistent and thus the NF's behavior correct in the face of dynamic scaling or failures is non-trivial.

There are several lines of work aiming at alleviating this problem. Generally, they can be classified in approaches that (i) keep all state local to a NF and transfer state when required [139, 152, 163], (ii) mix local and remote state [60, 153], and (iii) use centralized or distributed remote state [89, 188]. Relatable to SwingState [118] in this context is StateAlyzr [95], a static analysis framework for data plane programs. Given network function code, it identifies state that would need to be migrated and cloned to ensure state consistency in the face of traffic redistribution or failure. The authors find that for many network functions, their system can reduce the amount of state that needs to be migrated significantly compared to naive solutions.

Instead of continuously migrating state, we believe that the conceptually simple approaches around state externalization enabled through novel extremely low-latency interconnects, advanced caching, and failover strategies are promising. StatelessNF [89] is a prominent example of this approach leveraging the RAMCloud key-value store and InfiniBand networking.

4.3 Programmable Parsers

Perhaps the most fundamental operation of every network device is to parse packet headers to decide how packets should be processed. For example, a router uses the IP destination address to decide where to send a packet next and a firewall compares several fields against an access control list to decide whether to drop a packet. Packet parsing can be one of the main bottlenecks in high-speed networks because of the complexity of packet headers [62]. Packets have different lengths and consist of several levels of headers prepended to the packet payload. At each step of encapsulation, an identifier indicates the type of the next header or, eventually, the type of data subsequent to the header leading to long sequential dependencies in the parsing process.

Implementing low-latency parsers for high-speed networks is particularly challenging. To minimize overheads, switches often employ a *unified packet parser*. Such parsers use an algorithm that parses all supported packet header fields in a single pass. While this can improve performance, it also increases complexity and could become a security issue, especially for virtual switches [180].

Programmability is another key requirement as header formats may change over time, for instance due to new standards or the desire to support custom headers. Examples of more recent header structures include PBB, VxLAN, NVGRE, STT, or OTV, among many more. To support new or evolving protocols, a programmable parser can use a parse graph that is specified at runtime (e.g., leveraging state tables implemented in RAM and/or TCAM [62]).

4.4 Programmable Schedulers

Exposing programmable interfaces for scheduling and queuing strategies is another core functionality in the context of programmable networks. Sivaraman et al. [173] present a solution that allows known and future scheduling algorithms to be programmed into a switch without requiring hardware redesign. The proposed design uses the property that scheduling algorithms make two decisions: in which order and when to schedule packets. Additionally, the authors exploit the fact that in many scheduling algorithms a definitive decision on these two questions can be made at an early stage of processing: when a packet is enqueued. The resulting design uses a single abstraction: the **Push-In-First-Out queue (PIFO)**, a priority queue that maintains the scheduling order or time. Another design for a programmable packet scheduler was presented by Mittal et al. [129]. The authors show that while it is impossible to design a universal packet scheduling algorithm, the classic **Least Slack Time First (LSTF)** scheduling algorithm provides a sufficient approximation and can meet various network-wide objectives.

Implementing fair queuing mechanisms in high-speed switches is generally expensive, since complex flow classification, buffer allocation, and scheduling are required on a per-packet basis. Motivated by the question of how to achieve fair bandwidth allocation across all flows traversing

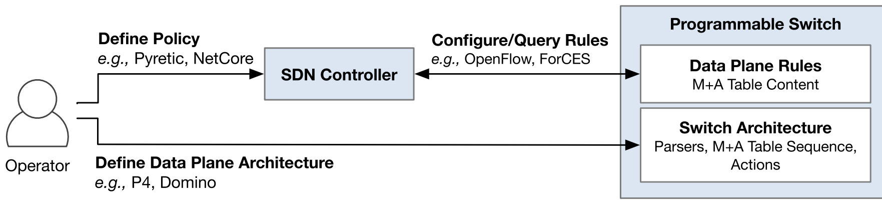


Fig. 5. Comparison of languages and protocols used in programmable data planes.

a link, Sharma et al. [168] present a dequeuing scheduler, called Rotating Strict Priority, which simulates an ideal round-robin scheme where each active flow transmits a single bit of data in every round. This allows the transmission of packets from multiple queues in approximately sorted order.

The trend toward increasing link speeds and slowdown in the scaling of CPU speeds leads to a situation where packet scheduling in software results in lower precision and higher CPU utilization. While this drawback can be overcome by offloading packet scheduling to hardware, doing so compromises the flexibility benefits of software packet schedulers. Ideally, packet scheduling in hardware should hence be programmable. Motivated by the insight that "in the era of hardware-accelerated computing, one should identify and offload common abstractions and primitives, rather than individual algorithms and protocols", Shrivastav in [169] proposes a generalization of the PIFO primitive used by state-of-the-art hardware packet schedulers: **Push-In-Extract-Out (PIEO)** maintains an ordered list of elements, but allows dequeueing from arbitrary positions in the list by supporting programmable predicate-based filtering when dequeuing. PIEO supports most scheduling (work-conserving and non-work conserving) algorithms that can be abstracted as the following scheduling policy: assign each element (packet/flow) an eligibility predicate and a rank. Whenever the link is idle, among all elements whose predicates are true, schedule the one with the smallest rank. The predicate determines when an element becomes eligible for scheduling, while rank decides the order amongst the eligible elements. The hardware design of the PIEO scheduler, also presented in Reference [169], demonstrates the scalability of this approach.

4.5 Programming Languages and Compilers

Over the last years, we have witnessed several promising efforts that go beyond low-level SDN protocols, such as OpenFlow, ForCES, or NETCONF. New high-level data plane programming languages allow to specify packet processing policies within a specific switch architecture in terms of abstract, generic, and modular language constructs. These efforts are largely driven by the needs of operators toward more complex SDN applications. Furthermore, the capabilities of modern, more flexible and programmable line rate networking hardware has motivated language approaches to specify the switch processing architecture (i.e., the layout of match-action tables and protocols supported in the parsing stage). The conceptual differences between these two classes of language abstractions found in programmable data plane systems today are depicted in Figure 5.

4.5.1 SDN Policy Definition. Languages for SDN programming generally differ in the amount of visibility that should be provided in SDNs (see Reference [41] for a discussion on this). A well-known language is Frenetic, a programming language for writing composable SDN applications using a set of high-level topology and packet-processing abstractions. Pyretic [56] improves on Frenetic by adding support for sequential composition, more advanced topology abstractions, and an abstract packet model that introduces virtual fields into packets. Modular applications can be written using the static policy language NetCore [132, 133], which provides primitive actions,

matching predicates, and query policies. Maple [185] simplifies SDN programming (i) by allowing a programmer to use a standard programming language to design an arbitrary, centralized algorithm, controlling the behavior of the entire network, and (ii) by providing an abstraction where the programmer-defined, centralized policy is applied to every packet entering a network.

Providing solid mathematical foundations to networking is one of the basic desires of SDNs. NetKAT [7] is one of the major efforts towards this objective. NetKAT proposes primitives for filtering, modifying, and transmitting packets, operators for combining programs in parallel and in sequence, and a Kleene star operator for iteration. NetKAT comes with provable guarantees that the language is sound and complete. In general, functional languages have become popular to provide such higher levels of abstractions, also including languages such as PFQ-Lang [20], which allows to exploit multi-queue NICs and multi-core architectures.

4.5.2 Low-level Data Plane Definition. At the heart of today's programmable data planes lies the question of how to specify and reconfigure the low-level architecture and configuration of programmable switching chips (i.e., the layout and sequence of match-action tables, the protocols understood by the protocol parser, and the actions supported) in an expressive and flexible manner.

An early and the most prominent language abstraction and compiler for specifying low-level packet processing functionality within programmable data planes is P4 [22]. Motivated by the limitations of existing SDN control protocols, such as OpenFlow, which only allow for a fixed set of header fields and actions, P4 makes it possible to define packet processing pipelines together with parsers and deparsers, match-action tables, and low-level operations that are applied to each packet. This language abstraction allows for protocol-independent packet processing by matching on arbitrary bit ranges and applying user-defined actions. Such abstract P4 programs are compiled for the specific underlying data plane target. The origins of P4 go back to work by Lavanya et al. [88] who study how to map logical lookup tables to physical ones while meeting data and control dependencies in the program. The authors also present algorithms to generate programs optimized for latency, pipeline occupancy, or power consumption. The compiled data plane program is then used to configure the underlying hardware or software target, and the P4-defined match-action tables are populated at runtime via a control interface, such as P4Runtime [34].

P4 rapidly gained immense popularity in the research community and is used in countless projects. Particularly, the wide range of supported targets from software switches to full reconfigurable ASICs as well as strong industry adoption make P4 a key enabling technology for comprehensive and flexible data plane programmability. For example, P4FPGA [186] is an open source compiler and runtime for P4 programs on FPGAs. By combining high-level programming abstractions offered by P4 with a flexible and powerful hardware target, P4FPGA allows developers to rapidly prototype and deploy new data plane applications. A second work in this direction is P4→NetFPGA [78], which integrates the function described with P4 in the NetFPGA processing pipeline. Other compilers exist for different software switching architectures, SmartNICs, and reconfigurable ASICs.

Extended programmability in the data plane also opens avenues for introducing bugs or writing insecure code. Ensuring correctness of programs is therefore also of high importance for data plane programs. Network verification and program analysis approaches aim at alleviating these issues. While widely in use in traditional network paradigms, network verification for fully programmable data plane systems is still an area of ongoing research. To this end, Dumitrescu et al. [48] propose a new tool and algorithm, called netdiff, to check the equivalence of related P4 programs and FIB updates to detect inconsistent behavior and bugs in data plane implementations. Also with the goal of simplifying P4 development, better testing programs, and identifying bugs early, Bai et al. propose NS-4 [10] a comprehensive simulation framework for P4-defined data planes. NS-4

integrates with the popular network simulator NS-3 and can efficiently simulate large multi-node networks running data planes written in P4.

While P4-like language abstractions dominate the programmable packet processing landscape, parts of the abstraction, in particular as required for stateful processing and scheduling, have not yet found a definitive standard. It appears that a single abstraction cannot cover all of the relevant aspects, and rather multiple pointed and specialized abstractions will emerge. While these subdomains are still being actively researched, we see the composition of the different abstractions as a major challenge for future research; the **Protocol-Independent Switch Architecture (PISA)** is a solid starting point in this space.

Takeaways — As discussed in this section, given their underlying complexity, programmable data plane systems are realized through a variety of abstractions. Some of them have been used in the networking domain for decades (e.g., the data flow or match table abstraction), while newer, more general, and often language-based abstractions for custom packet actions, stateful packet processing, and programmable parsing are mostly motivated by new hardware capabilities. In particular, these newer programming models are still rather inflexible and cumbersome. Going forward, akin to high-level synthesis approaches, we expect to see more unified data plane programming abstractions at higher levels that cover the full spectrum from P4-like programming for ASICs to eBPF or C-like programming for CPUs. Similarly, it remains unanswered how different, independent data plane programs should run alongside on the same hardware. This is required for modular composition of network programs, and may eventually enable multi-tenant virtualization scenarios.

5 ALGORITHMS AND HARDWARE REALIZATIONS

Data planes rely on various algorithms and data structures for packet processing, often to be implemented in hardware. We will now discuss the most relevant work in this area.

5.1 Reconfigurable Match-action Tables

Traditional OpenFlow hardware switch implementations allow packet processing on a fixed set of fields only. Reconfigurable match tables, such as RMT [23], allow the programmer to match on and modify all header fields (or arbitrary bit ranges) making the devices significantly more flexible and capable. RMT, for example, is a RISC-inspired pipelined architecture for switching chips that provides a minimal set of action primitives to specify how headers are processed in hardware. This makes it possible to change the forwarding plane without requiring new hardware designs.

5.1.1 Exact Matching Tables. Large networks (such as those in data centers running millions of VMs) require efficient algorithms and data structures for their **Forwarding Information Bases (FIB)** to that scale to millions of entries on commodity switching chips. An attractive approach to realize such memory-efficient and fast exact match FIB operations in software switches is to employ highly concurrent *hash tables*. For example, solutions based on cuckoo hashing, such as CuckooSwitch [194], have been able to process high packet rates across the PCI bus of the underlying hardware while maintaining a forwarding table of one billion forwarding entries.

5.1.2 Prefix Matching Tables. Programmable switches implementing match-action tables in hardware generally need to support different types of operations and tables. Besides exact matches, especially IP address lookups and prefix matching are frequent operations and have thus received much attention in the research community. Given the heavily constrained resources on devices, besides optimizing lookup time, it is important to improve memory efficiency of match-action table representations in hardware. A natural solution to improve the memory

efficiency of IP forwarding tables is to employ *FIB aggregation*, by replacing the existing set of rules by an equivalent but smaller representation. Such aggregations can either be performed statically (such as ORTC [47]) or dynamically (such as FIFA [116], SMALTA [183], or SAIL [191]). Rétvári et al. [155] explored the application of compressed data structures to reduce FIB table sizes to an information-theoretical optimum without sacrificing the efficiency of standard operations such as longest prefix match and FIB update. An implementation of their approach in the Linux kernel (using a re-design of the IP prefix tree) shows the feasibility and benefits of this approach.

Inspired by Zipf's law, the empirical fact that certain rules are used much more frequently than others, caching represents another optimization opportunity. For instance, it may be sufficient to cache only a small fraction of the rules on the fast expensive hardware fast path; less frequently used rules can then be moved to less expensive storage (e.g., to the DRAM of the route processor or software-defined controller). Different FIB caching schemes use different algorithms that minimize the number of updates needed to the cache [17, 18].

In the context of virtual routers used for flexible network services such as customer-specific and policy-based routing, further challenges related to resource constraints arise. In particular, supporting separate FIBs for each virtual router can lead to significant memory scaling problems. Fu et al. [58] propose using a shared data structure and a fast lookup algorithm that capitalizes on the commonality of IP prefixes between virtual FIB instances.

5.1.3 Wildcard Packet Classification. Packet classification, the core mechanism that enables networking services such as firewall packet filtering and traffic accounting, is typically either implemented using ternary TCAMs or software. Both TCAM and software-based approaches usually entail tradeoffs between (memory) space and (lookup) time. **Content-addressable memory (CAM)** and **Ternary CAM (TCAM)** chips are the most important components in programmable switch ASICs to perform packet classification on configurable header fields. Using dedicated circuitry, rules can be matched in priority order and in only a single clock cycle. In particular, TCAMs classify packets in constant time by comparing a packet with all classification rules of ternary encoding in parallel.

A major design challenge of large-capacity CAMs is to reduce power consumption associated with the vast amount of parallel active circuitry without sacrificing speed or memory density, and while supporting multidimensional lookup. Despite their high speed, TCAMs can also suffer from a range expansion problem. When packet classification rules have fields specified as ranges, converting such rules to TCAM-compatible rules may result in an explosion of the number of rules.

One approach to reduce TCAM power consumption for high-dimensional classification is to employ pre-classifiers (e.g., considering just two fields such as the source and destination IP addresses). The high-dimensional problem can thereby use only a small portion of a TCAM for a given packet. Ma et al. [120] show how to design a pre-classifier such that a packet matches at most one entry in the pre-classifier, avoiding rule replication. SAX-PAC in turn exploits the observation that most practical classifiers include many independent rules, allowing for matching in arbitrary order and only considering a small subset of dimensions [98]. Furthermore, TCAM Razor [111] strives to generate a semantically equivalent packet classifier that requires the least number of TCAM entries. The negative space-time tradeoff, which seems inherent in the design of classifiers, can sometimes be overcome allowing for range constraints, among others [98].

Perhaps the most prominent application of generic wildcard packet classifiers, the Open vSwitch fast-path packet classifier [145] uses a combination of extensive multi-level hierarchical flow-caching and the venerable **Tuple Space Search scheme (TSS)** [176]. TSS exploits the observation that real rule databases typically use only a small number of distinct field lengths, therefore, by mapping rules to tuples, even a simple linear search of the tuple space can provide significant

speedup over a naive linear search over the filters. In TSS, each tuple is maintained in a hash table that can be searched in constant time. Though TSS is used extensively in practice, recently, it has been shown that the linear search phase can be exploited in a malicious algorithmic complexity attack to exhaust data plane resources and launch a denial of service attack [40].

5.2 Fast Table Updates

Match-action tables should not only support a fast lookup but also fast updates for inserting, modifying, or deleting rules. Such updates can be accelerated by partitioning and optimizing the TCAM. For example, Hermes [31] trades a nominal amount of TCAM space for assuring improved performance. Also, a hybrid software-hardware switch, such as ShadowSwitch [19], can help lower the flow table entry installation time. Since software tables can be updated very fast, table updates should happen in software first before being propagated to TCAM to offload software forwarding and achieve higher overall throughput. Lookups in software should only be performed in case there are no entries matching a packet in hardware. Solutions such as ShadowSwitch further exploit the fact that deleting TCAM entries is much faster than adding them, proposing translating adding entries to a mix of adding in software tables and deleting from hardware tables.

Takeaways — In general, as the network data plane becomes increasingly programmable and includes more and more embedded algorithms and data structures, research on efficient and dependable approaches will remain active in coming years. Especially the network data plane within cloud environments is immensely complex already today and maintains substantial embedded state; ubiquitous virtualization may increase complexity even further in the future. Since cloud resources also allow shared access and configuration from tenants, future research on reliable and available algorithms and data structures for cloud data planes is crucial. We believe that the emergence of new types of attacks, such as algorithmic complexity attacks, demand data plane algorithms to provide hard real-time constraints on the amount of resources used for a specific task; the latter is especially important for resource-constrained devices. In addition to complexity, due to quickly growing traffic rates, also scalability of these algorithms remains an important open problem.

6 APPLICATIONS

The appearance of programmable data planes has started a trend toward moving certain general information-processing functionality, formerly implemented either entirely in software or on dedicated hardware appliances, directly into the network data plane. The ability to program network devices suddenly changes a *dumb* pipe that only moves data into a complete, sophisticated data processing pipeline that is able to transform data as it flows. Applications that have been offloaded to the network in this manner include telemetry, massive-scale data processing, machine learning, and even complete key-value stores. Network devices already sit in the data path and, as a result, offloading additional functionality here minimizes the need for additional, potentially expensive, data movement and reduces the end-to-end processing latency. In addition, many applications may benefit from the new visibility into the network (e.g., queue occupancy levels) or from the energy savings possible by running conventional compute tasks on low-power programmable NICs [114].

One may wonder which types of applications may benefit most from being offloaded into the programmable data plane [160]. Is there an over-arching scheme that would help identify when to consider the data plane implementation for a particular use case? Judging from recent examples, we see that the typical applications are the ones that (i) *process massive amounts of network-bound data* or have a strong networking component in some way (e.g., implement request-response patterns), (ii) *pose stringent latency and/or throughput requirements*, and (iii) *can be decomposed into a small*

set of simple primitives that lend themselves readily to be implemented partially or entirely on top of packet processing primitives exposed by programmable data plane devices.

Below, we highlight some of the well-known examples for data plane offloading from the literature, including virtual switching, in-network computation, telemetry, distributed consensus, resilient and efficient forwarding, and load balancing.

6.1 Monitoring, Telemetry, and Measurement

Perhaps the most interesting applications for data plane offloading are related to network measurement, telemetry, monitoring, and diagnosis. This is mostly because these applications share traits that make them particularly suitable for data plane-based implementations: They operate at massive traffic scale and under tight performance requirements. Most importantly, the data plane has direct and low-latency access to monitoring application's input data, that is, network packets or measurements taken in the data plane. For decades, the state-of-the-art has required mirroring monitored traffic to dedicated middleboxes, involving costly traffic duplication and software processing; consequently, the efficiency gains with in-network data plane implementations can be enormous. We therefore see programmable data planes as a game changer in this context, providing deep insights into the network, even to end hosts, as we discuss in the following.

At the heart of many approaches lies the goal to improve the visibility into network behavior. Jeyakumar et al. [83] present a solution that not only provides improved visibility to end hosts but also allows to quickly introduce new data plane functionality, via a new **Tiny Packet Program (TPP)** interface. Rooted in the work on Smart Packets [162] originally proposed for on-switch network management and monitoring based on the Active Network paradigm [52], TPPs are embedded into packets by end hosts and can actively query and manipulate internal network state. The approach is based on the "division of labor" principle: Switches forward and execute TPPs in-band at line rate and end hosts perform flexible computation on the network state exposed by the TPPs. The authors also present a number of use cases motivating in-band network telemetry. The general framework for **in-band network telemetry (INT)** was later presented by Kim et al. in Reference [96].

As a step toward generalized measurement, one direction of work has looked at sketches as a new data plane structure for network analytics. Sketches, which leverage probabilistic, sub-linear data structures, are an efficient way to maintain summarizing statistics and metrics over large input datasets [6]. OpenSketch [192] provides a library of such sketches while UnivMon [117] introduces a universal streaming scheme, where a generic sketch in hardware preprocesses packet records at high rates and software applications compute application-specific metrics. Recently, SketchVisor [77] presented a comprehensive network measurement framework that augments sketch-based measurement in the data plane with a fast path that is activated under high traffic load to provide high-performance, local measurement with slight degradation in accuracy.

To make network monitoring systems more flexible, researchers have sought ways that allow network operators to write network measurement queries directly and in a more expressive way, instead of relying on a particular sketch. These queries can then be compiled to run on modern programmable switches at line rate. Marple [135] identifies a set of fixed operators that can be compiled to programmable hardware and used to compose a wide range of network monitoring queries. This approach offers great performance for any analytics tasks that can fit entirely in a programmable switch, but it also requires software offload once the device's SRAM and ALU resources are full. Sonata [69] improves on this hardware-restrictive model by more intelligently dividing a query into parts that are executed on the switch and parts that are executed on a general-purpose software stream processor. Motivated by the limited processing capabilities of software stream processing systems, Sonata introduces a method of iterative refinement that can reduce

the amount of traffic sent to software. This iterative refinement, however, comes at the cost of using significant SRAM and ALU resources on the switch. It also requires relaxing the temporal and logical constraints of a query.

Further applications of in-network measurement are related to heavy hitter detection [149, 174], traffic matrix estimation [64], and TCP performance measurements [61]. First, HashPipe [174] realizes heavy-hitter detection entirely in the data plane. HashPipe implements a pipeline of hash tables, which retain counters for heavy flows while evicting lighter flows over time. Second, Gong et al. [64] show that by designing feasible traffic measurement rules (installed in TCAM entries of SDN switches) and collecting the statistics of these rules, fine-grained estimates of the traffic matrix are also possible. Finally, Dapper [61] allows for analyzing TCP performance problems in real time right near the end-hosts, i.e., at the hypervisor, NIC, or top-of-rack switch. This makes it possible for the operator to determine whether a particular connection is limited by the sender, the network, or the receiver, and to intervene accordingly in a timely manner.

Finally, an orthogonal line of work identifies that programmable switches, while not suitable for practical and ubiquitous offload of analytics tasks due to resource constraints, are useful for accelerating and enhancing telemetry systems. Instead of compiling entire queries to a programmable switch, *Flow [175] places parts of the select and grouping logic that is common to all queries into a hardware match-action pipeline. In *Flow, programmable line rate switches export a stream of *grouped packet vectors* **(GPVs)** to software processors. A GPV contains a flow key (e.g., an IP 5-tuple) and a variable-length list of packet feature tuples (e.g., timestamps and sizes) from a sequence of packets in that flow. GPVs are generated through a novel in-network key-value cache that can be implemented as a sequence of match-action tables for programmable switches. The authors expanded on the telemetry system with high-performance network analytics platform [128].

Sketches and entirely switch-based approaches to monitoring and telemetry provide unprecedented performance for simple counters and basic queries. Besides requiring significant amounts of scarce switch resources and imposing operational inflexibilities, these approaches lack the packet-level granularity that modern fine-grained network analytics solutions require. We therefore see large potential in hybrid approaches leveraging both high-performance switch-based telemetry together with flexible software-based analytics as proposed in Sonata [69] and *Flow [175]. Finding the right balance between in-network and host processing, taking into account novel processing platforms such as FPGAs, will remain a hot topic for years to come.

6.2 Virtual Switching

Virtual networking is heavily used in data centers and cloud computing infrastructure. At the heart of cloud computing lie the ideas of resource sharing and *multi-tenancy*: independent instances (e.g., applications or tenants) can concurrently utilize the physical infrastructure including their compute, storage, and management resources [100]. While physically integrated, network virtualization enables logical isolation of resources for each tenant. *Virtual switches* are a core network component in this architecture located in the virtualization layer of servers connecting tenants' host-based compute and storage resources among each other and to the rest of the network [81, 100, 142].

Using *flow table-level isolation*, the flow tables in the virtual switch are divided into per-tenant logical data paths that are populated with sufficient flow table entries to link the tenants' resources into a common interconnected workspace [81, 100, 142]. Practically, this workspace is an overlay network realized through a tunneling protocol, such as VXLAN.

Despite the widespread deployment of virtual networking [42, 54, 87], providing sufficient (logical and performance) isolation remains a key challenge. Serious isolation problems with the **Open vSwitch (OVS)** [145] have been reported in Reference [181]: An adversary could not only

break out of the VM and attack all applications on the host, but could also manifest as a worm compromising an entire data center. Other severe isolation vulnerabilities, also in OVS, enable cross-tenant denial-of-service attacks [40]. Such attacks may exacerbate concerns over the security and adoption of public clouds. Jin et al. [84] are the first to point out security weaknesses of co-locating virtual switches with the hypervisor, proposing stronger isolation mechanisms. In response, MTS [179] proposes placing per-tenant virtual switches in VMs for increased security isolation.

As an alternative to the host-based virtual switch model, implementing virtual networking can also be offloaded to the NIC. While commodity NICs have basic support for switching among virtual machines through SR-IOV and offloads of standard tunneling protocols used in this context, such as VXLAN and NVGRE [65], programmability at the network edge is invaluable for implementing custom virtualization solutions. While this is already possible in software on platforms like OVS, having a similar level of programmability on NICs can significantly enhance scalability and lower cost of virtualization in data centers. AccelNet [54] is an early example of employing such an architecture, which we expect will become standard practice going forward.

6.3 In-network Computation

In-network computation is a promising way to address performance bottlenecks and scalability limits of massive network-bound data processing in data centers as often performed in machine learning and big data processing frameworks [1, 46]. Such analytics, graph processing, and learning applications, to name a few, exhibit a few characteristic communication patterns that make them suitable for (partial) implementations in the data plane. First, they usually substantially reduce and aggregate the data during processing (e.g., take the sum of the inputs or find the minimum). It is therefore beneficial to apply these functions as early as possible to decrease the amount of network traffic and reduce congestion. Second, they are usually characterized by simple arithmetic/logic operations that make them suitable for massive parallelization and execution on programmable hardware. Third, in many algorithms these operations are also commutative and associative implying that they can be applied separately and in arbitrary order on different portions of the input data without affecting the correctness of the end result.

Correspondingly, most big data applications follow the *map-reduce* pattern to achieve massive horizontal scaling: Large-scale computation instances are first partitioned across many edge servers that do partial processing on smaller chunks before the results are again aggregated to obtain the final result. Such many-to-few communication patterns (often referred to as *incast*) are, however, poorly supported in data center deployments incurring significant performance issues.

The first attempt at departing from performing data aggregation at edge servers is Camdoop [39], which supports on-path aggregation for map-reduce applications on top of a direct-connect data center fabric where all traffic is forwarded between servers without switches. While this significantly reduces network traffic and provides a performance increase, it requires a custom network topology and is incompatible with common data center infrastructure. Netagg [121] was a proposal to avoid the limitations of Camdoop by implementing on-path aggregation inside the network layer at dedicated middleboxes. Netagg improves job completion times significantly across a wide range of big data workloads and frameworks including Apache Hadoop. Later, SHArP [67] removed dedicated "network accelerator" middleboxes from the in-network computation stack and presented a generic programmable data plane hardware architecture for efficient data reduction, relying on scalable in-network trees and pipelining to reduce latency for big data processing.

Toward the generalization of these approaches, Liu et al. lay the foundations of an in-network computation framework by presenting a minimal set of abstractions they call IncBricks [114]: an in-network caching fabric with basic computing primitives based on programmable network

devices. The authors in Reference [167] furthermore ask the related general question of how to overcome the limitations imposed by the usually scarce resources provided on programmable switches, such as limited state storage and limited types of operations, for in-network computation tasks. They identify general building blocks that can be used to mask these limitations of programmable switches using approximation techniques and then implement several approximate variants of congestion control and load balancing protocols, such as XCP, RCP, and CONGA [5] that require explicit support from the network.

Going even further, the most recent innovations in in-network computation are based on the observation that the network itself may also be used as an accelerator for workloads that are (at first sight) unrelated to networking or packet processing. In particular, machine learning and artificial intelligence workloads have emerged as promising candidates to be (partially) implemented within the network [159]. More specifically, programmable network devices may be a suitable engine for implementing a CPU's Artificial Neural Networks co-processor. N2Net [171] is an example of an in-network neural network, based on commodity switching chips deployed in network switches and routers. Another interesting application that can be implemented in the network is string matching for accelerating information retrieval and language processing use cases. PPS [82] is an in-network string matching implementation for programmable switches. The PPS compiler translates a set of keywords to **Deterministic Finite Automata (DFA)** that can then be realized in hardware as a sequence of match-action tables yielding significantly higher matching throughput than comparable software implementations.

These and other advances in leveraging the network itself as a compute platform for a wide variety of workloads demonstrates the versatility and potential of programmable data planes. It is too early to tell which (not directly networking-related) workloads we will see being offloaded to the network ubiquitously and which applications will remain more illustrative and experimental. Nevertheless, given scalability limitations of general-purpose compute resources, we anticipate architectures leveraging in-network computation to be transformative for many workloads.

6.4 Distributed Consensus

Perhaps viewable as a special case of in-network computation, distributed consensus deserves special discourse not only because of the substantial research treatment that it received over the past years but also because it exhibits a special network requirement profile: While general in-network computation is mostly throughput-bound, distributed consensus is much more latency-oriented, often posing delay requirements on the order of a single server-to-server round-trip time (or even less; see Reference [85]). Distributed consensus describes the coordination among controllers or switches to perform a computation jointly and reliably, even in the presence of network failures, arbitrary communication delays, or Byzantine participants. Applications include leader selection, clock synchronization, state replication, and general multi-write key-value stores.

NetPaxos [44] demonstrates the feasibility of implementing the venerable Paxos distributed consensus protocol [103] in network devices, either using certain OpenFlow extensions or by making some assumptions about how the network orders messages. Although neither of these protocols can be fully implemented without changes to the underlying switch firmware, the authors argue that such changes are feasible in existing hardware. Dang et al. [43] also show the performance benefits achievable by offloading Paxos into the data plane and describe an implementation in P4.

In-band mechanisms in the data plane can also be used for synchronization and coordination of other distributed systems components, such as SDN controllers. Schiff et al. [161] propose a synchronization framework based on atomic transactions implemented on switches and show that this approach allows realizing fundamental consensus primitives in the presence of failures.

In the context of data centers, NetChain [85] provides scale-free coordination within a single server-to-server **round trip time (RTT)**, or even less (half of an RTT!). This is achieved by allowing programmable switches to store data and process queries entirely in the data plane, which eliminates the query processing at coordination servers and cuts the end-to-end latency perceived by clients to as little as the processing delay from their own software stack plus network delay. NetChain relies on new protocols and algorithms guaranteeing strong consistency and switch failure handling. Extending these principles to key-value stores, NetCache [86] implements a small key-value store cache in a programmable hardware switch. The switch works as a cache at the data center's rack-level, handling requests directed to the rack's servers. The implementation deals with consistency problems and shows how to overcome the constraints of hardware to provide throughput and latency improvements. SwitchKV [109] generalizes this idea by implementing a generic data plane-based key-value query accelerator, with significant improvements in throughput and latency. Programmable network switches act as fast key-value caches by keeping track of cached keys and routing requests at line speed based on the query keys encoded in packet headers, so the data plane cache nodes absorb the hottest queries and, therefore, no individual key-value store backend server is overloaded. Furthermore, specialized in-switch key-value stores for network measurement collection and aggregation appear in *Flow [175], Marple [135], and IncBricks [114].

Perhaps an unlikely place to find distributed consensus protocols is in the programmable devices themselves. Deep inside a typical programmable switch lies a rather complex distributed appliance, with multiple match-action tables, parsers, queues, and so on, closely cooperating to perform consistent and fast packet processing. It turns out that consistently applying modifications to this pipeline is a rather complex task, in sore need of strong consistency guarantees. BlueSwitch [70] has presented a programmable network hardware design that supports a transactional packet-consistent configuration mechanism: All packets traversing the data path will encounter either the old or the new configuration, and never an inconsistent mix of the two. This will help avoid network transients like blackholes and micro-loops that often plague today's networks [66].

6.5 Resilient, Robust, and Efficient Forwarding

Data planes operate at much faster pace than the typical control plane usually implemented in software. This motivates moving functionality for maintaining connectivity under failures into the switches. At the same time, offloading control planes is non-trivial.

The authors in Reference [41] make the observation that typical SDN workloads impose significant communication overheads due to frequent interaction between the control and data plane. Some of the control plane functionality can, however, be efficiently offloaded from the controller to the switch itself. To meet the needs of high-performance networks, the authors propose and evaluate DevoFlow, a modification of the OpenFlow model that breaks the tight coupling between the SDN control plane and the data plane in a way that maintains a useful amount of visibility for the former without imposing unnecessary communication costs. For common SDN applications, DevoFlow requires notably fewer flow table entries and results in reduced controller-switch communication compared to a traditional OpenFlow realization. Molero et al. [130] take this idea further and make a general case for offloading control plane protocols entirely to the data plane. Motivated by long convergence times of traditional routing protocols, the authors show that modern programmable switches are powerful enough to run many control plane tasks directly in hardware. As a proof of concept, the authors implement a path vector protocol for programmable data planes in P4 that rapidly converges in the case of link failure while fully respecting BGP-like routing policies.

The design of resilient data planes has been studied intensively in the literature. To provide high availability, connectivity, and robustness, dependable networks must implement functionality for

in-band network traversals (e.g., to find failover paths in the presence link failures [21]). Here, mechanisms based on dynamic state at the switches provide interesting advantages compared to simple stateless mechanisms or mechanisms based on packet tagging. Liu et al. [112] propose transferring responsibility for maintaining basic network connectivity entirely into the data plane, which operates much faster than the control plane. Their approach to ensure connectivity via data plane mechanisms relies on link reversal routing, adapted to handle operational concerns such as message loss or arbitrary delay from the original algorithm by Gafni and Bertsekas [59] (see also Reference [141]). Holterbach et al. [75] provide an implementation for automatic data-driven fast reroute entirely in the data plane. Their system, Blink, runs on programmable line-rate switches and detects remote outages by analyzing TCP behavior directly within the switch. In case of failure, Blink quickly restores connectivity and reroutes traffic without control plane involvement.

While offloading control plane functionality contradicts one of the core concepts of SDN, namely, reducing the complexity of the data plane by having simple forwarding functions, data plane programmability enables flexibility to the operator in what functions are performed in the network directly. This is opposed to and much more cost-effective than the traditional approach of making network devices generally *smarter* by embedding complex functionality into the data plane by default, which in turn increases overall complexity. We believe that finding the right balance between control plane and data plane responsibilities will remain a hot topic for the years to come.

6.6 Load Balancing

Related to resilient routing, programmable data planes provide unprecedented flexibilities and performance in how traffic can be dynamically load balanced across multiple forwarding paths, workers, or backend servers. For instance, Hedera [3] can also be viewed as a load balancer. The aim is to implement the "resource pooling" principle using horizontal scaling [187], making a collection of independent resources behave like a single pooled resource to exploit statistical multiplexing, load distribution, and improved failure resilience.

A well-known example is HULA [93], a scalable load balancing solution using programmable data planes. HULA is motivated by the shortcomings of ECMP routing as well as of existing congestion-aware load balancing techniques such as CONGA [5]. Due to limited switch memory, these approaches can only maintain a subset of congestion-tracking state at the edge switches and hence do not scale. HULA is flexible and scalable as each switch tracks congestion only for the best path to a destination through a neighboring switch. Another example of a load balancing application is SilkRoad [126], which leverages programmable ASICs to build faster load balancers.

Beyond multi-path load balancers, MBalancer [25] addresses the load balancing problem in the context of key-value stores. In particular, distributed key-value stores often have to deal with highly skewed key-popularity distributions, making it difficult to balance load across multiple backends. MBalancer is a switch-based L7 load balancing scheme, which offloads requests from bottleneck Memcached servers by identifying hot keys in the data plane, duplicating these hot keys to multiple Memcached servers, and then adjusting the switches' forwarding tables accordingly.

Takeaways — Throughout this section, we have explored a multitude of applications leveraging programmable data plane technology. We can observe that use cases that have been around for some time, such as network monitoring or virtual switching, are becoming hot research topics again. Data plane programmability opens avenues to realize these applications at scale and granularity that was previously either impossible or prohibitively expensive. While (so far) the greatest benefits appear for applications that mainly revolve around networking tasks, we see an increasing number of applications from other (albeit network-related) domains to benefit from data plane

programmability, including the Internet-of-Things [14] and wireless and mobile networks [190]. More generalized in-network computation is still in its infancy, and we expect to see more research in the direction of offloading applications from various domains to programmable data plane devices. Many of those applications mostly reside at the edge of the network and in the end hosts. This is aligned with a general trend in the research community where programmable data plane technology is increasingly employed at the host-network boundary [33]. In particular, accelerators placed at end hosts, such as SmartNICs, are a promising platform for this direction.

7 RESEARCH CHALLENGES

To sum up this survey and share our learnings, we provide a short discussion of major open issues and research challenges we see in this space.

Improved Abstractions *Which abstractions provide an optimal tradeoff between functionality, performance, and API simplicity?*

The art and science of programmable switch architectures revolve around abstractions. Ideally, an abstraction should be simple enough to capture just the right amount of configurable data plane functionality to admit efficient hardware and software implementations, but expressive enough to allow higher layers to synthesize complex packet processing behavior. Moreover, such an abstraction should be easily exposable to the control plane through a secure and efficient data plane API [34, 125]. It should adequately handle global state embedded in the data plane and provide a well-defined consistency model [188]. Also, it should admit analytic performance models [11, 131] and automatic program transformations for performance optimization [131]. It should separate static semantics from dynamic behavior [154]. And, last but not least, it should embrace a convenient mental model that is familiar to network operators and programmers.

Efficient Reconfigurability *How can we support more efficient yet consistent reconfigurability in the data plane?*

Alongside the move from the rigid programming model of OpenFlow to the more flexible P4 world is the desire to expose every aspect of processing functionality a switch may perform to be reconfigured for different and changing use cases in a flexible and efficient manner. This is not limited to the way packet processing policies are represented in the data plane, including the method by which packets are associated with the respective processing actions to be executed on them; it also extends to further critical packet processing operations and the reconfigurability thereof, ranging from programmable packet parsing [62] to universal scheduling and queuing schemes [129, 173]. In particular, changing data plane behavior at runtime without disrupting packet processing [175] remains an open problem.

Scalability *How can we realize high-performance data planes, especially stateful ones?*

The need to scale systems to handle massive workloads increasingly pushes designers to explore more complex solutions that handle application state already in the data plane [86, 126, 167]. While stateless packet processing approaches are rather solid at this point in time, stateful approaches are still in their infancy and no clear winner has emerged yet. The complexity of a stateful abstraction lays in the need to address state management problems (e.g., consistency) in a programmer-friendly way while guaranteeing high performance. This is especially challenging as frequently reading from and writing to memory, as it is continuously required in packet processing workloads, is still one of the main sources of performance issues in modern computing systems [23].

Network Automation *How can we design more self-adjusting networks that map high-level policies to the underlying physical infrastructure and autonomously adapt to changing demands or failures?*

A major current trend in networking concerns *automation*. Over the last years, the vision of "self-driving" communication networks that adapt and optimize themselves towards their current workload has emerged. Related to this trend is also the notion of "intent-based networking," which describes the vision of designing and operating networks in terms of higher-level business policies, and letting the network deal with low-level concerns in an automated, data-driven, agile, secure, and verifiable way [27]. Recent progress in high-level network programming languages has delivered important insights to realize the vision of intent-based networking in the form of efficient language constructs and modular composition frameworks [56, 88, 97, 133, 185, 193]. Yet, how to best expose data plane functionality to the operator offering the maximum programming freedom while masking the underlying complexities efficiently remains unclear. Ideally, an "intent-based data plane compiler" should actively attempt to find the data plane representation that would yield the highest performance [131] with the minimal data plane footprint [111, 155], built on a firm theoretical foundation for optimizing data plane programs and reasoning about performance [11, 131].

Verification, Monitoring, and Security *How can we design efficient verification, monitoring, and security frameworks that allow the operator to reliably reason about the correctness, performance, and security of the data plane?*

Data plane compilation, that is, downward mapping from the intent layer to the data plane, is just one side of the coin. In fact, highly related to the challenges associated with automatically adapting the network to changing environments is the need to verify the correctness and desired effect of a configuration change. To close the control loop, an upwards mapping is also necessary, which would permit the control plane to monitor and verify the operations of the data plane. Indeed, recent results indicate that the network should be architected from the ground up with verifiability and security in mind [97, 170], requiring new abstractions. Related to verifying correctness, as programmability also opens up more ways to introduce vulnerabilities and new attack surfaces, it is important to ensure that the data plane operates in a secure manner. While significant work has been done on the security of SDNs in general, we believe that new, extensively programmable data plane systems will require new security models and verification objectives. For example, many such attack vectors are related to compilers; fuzzing is a promising direction for uncovering such bugs [2, 158]. In general, given the mission-critical role the data plane plays, the success of novel data plane technologies will depend on the reliability and security guarantees they can provide.

8 CONCLUSION

Before concluding, we present a broad classification of the key papers discussed throughout this survey. This taxonomy is split between foundational contributions that enable data plane programmability (Figure 6) and works that leverage programmable data planes for novel use cases and applications (Figure 7). Additionally, as an annex to this survey, a reading list for students, practitioners, and researchers interested in programmable data planes is available online [127].

Motivated by the changing demands in packet processing toward flexibility, programmability, and high performance, novel ideas and solutions are needed to quickly and cost-efficiently support change. Programmable networks in general, and programmable data planes in particular, provide exactly that: an inexpensive alternative to supporting all possible packet processing functionality at once. Programmable networks also enable niche solutions: solutions that would not have been worthwhile for vendors due to the small-scale market. While greater flexibility through comprehensive programmability and reconfigurability benefits operators and vendors who wish to provide custom-tailored solutions and new use cases for clients, it also vastly increases the complexity

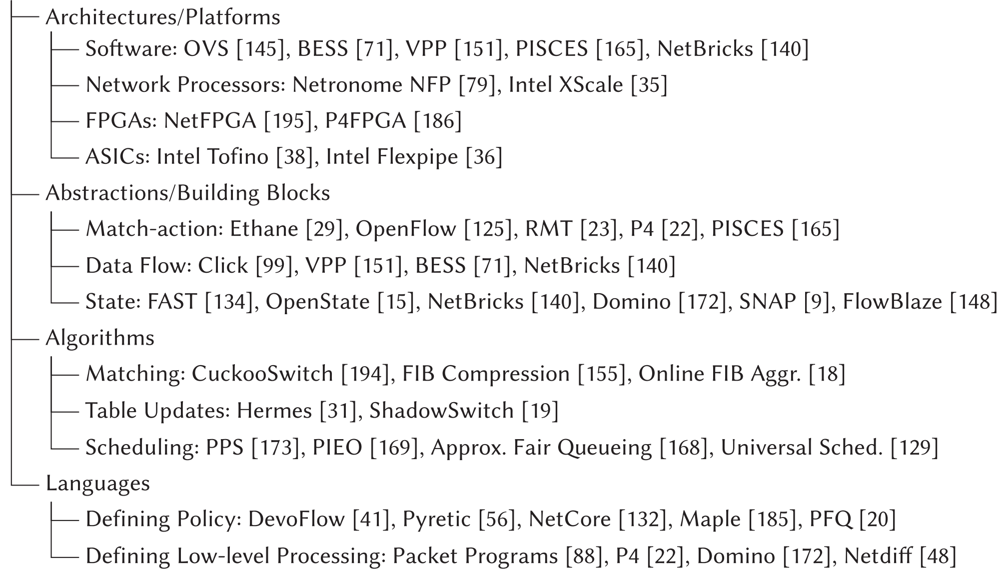


Fig. 6. Taxonomy of works laying the foundations of programmable data plane technology.

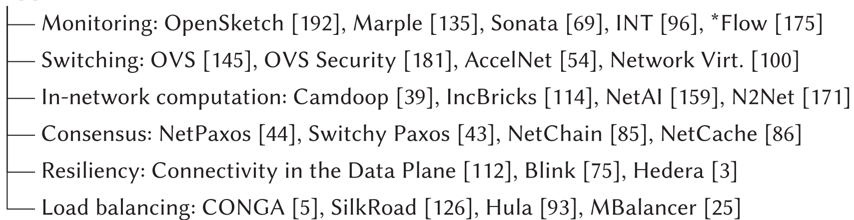


Fig. 7. Taxonomy of key applications built on top of programmable data planes.

of networking abstractions. Finally, low-level programmability introduces more ways to introduce bugs and vulnerabilities into the highly critical data plane. Apart from uncovering new use cases, accelerating existing applications, or enabling them at unprecedented scale, we see the largest fundamental challenges in providing powerful, universal abstractions with security and scalability in mind that span the vast array of available platforms, languages, and use cases.

While the body of existing work covered in this survey is already vast, we believe network programmability is still in its infancy. We expect that this rapidly evolving field will significantly affect the interfaces and interactions between applications and networks and, therefore, contribute to the design of future computer architectures.

ACKNOWLEDGMENT

The idea for this survey originated from discussions during the Dagstuhl Seminar 19141 on Programmable Network Data Planes [8].

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving et al. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI'16*. USENIX.

[2] Andrei Alexandru Agape, Madalin Claudiu Danceanu, Rene Rydhof Hansen, and Stefan Schmid. 2021. P4Fuzz: Compiler fuzzer for dependable programmable dataplanes. In *ICDCN'21*. ACM.
[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. 2010. Hedera: Dynamic flow scheduling for data center networks. In *NSDI'10*. USENIX.
[4] Omid Alipourfard and Minlan Yu. 2018. Decoupling algorithms and optimizations in network functions. In *HotNets'18*. ACM.
[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut et al. 2014. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM'14*. ACM.
[6] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The space complexity of approximating the frequency moments. In *STOC'96*. ACM.
[7] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. In *POPL'14*. ACM.
[8] Gianni Antichi, Theophilus Benson, Nate Foster, Fernando M. V. Ramos, and Justine Sherry. 2019. Programmable network data planes. *Dagstuhl Rep.* 9, 3 (2019), 178–201. DOI: https://doi.org/10.4230/DagRep.9.3.178
[9] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful network-wide abstractions for packet processing. In *SIGCOMM'16*. ACM.
[10] Jiasong Bai, Jun Bi, Peng Kuang et al. 2018. NS4: Enabling programmable data plane simulation. In *SOSR'18*. ACM.
[11] Manu Bansal, Aaron Schulman, and Sachin Katti. 2015. Atomix: A framework for deploying signal processing applications on wireless infrastructure. In *NSDI'15*. USENIX.
[12] David Barach, Leonardo Linguaglossa, Damjan Marion et al. 2018. High-speed software data plane via vectorized packet processing. *IEEE Commun.* 56, 12 (2018).
[13] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *ANCS'15*. IEEE.
[14] S. Bera, S. Misra, and A. V. Vasilakos. 2017. Software-defined networking for Internet of Things: A survey. *IEEE Internet Things J.* 4, 6 (2017).
[15] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. 2014. OpenState: Programming platform-independent stateful Openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.* 44, 2 (2014).
[16] Giuseppe Bianchi, Marco Bonola, Salvatore Pontarelli, Davide Sanvito, Antonio Capone, and Carmelo Cascone. 2016. Open Packet Processor: A programmable architecture for wire speed platform-independent stateful in-network processing. arxiv:1605.01977.
[17] Marcin Bienkowski, Jan Marcinkowski, Maciej Pacut, Stefan Schmid, and Aleksandra Spyra. 2017. Online tree caching. In *SPAA'17*. ACM.
[18] Marcin Bienkowski, Nadi Sarrar, Stefan Schmid, and Steve Uhlig. 2018. Online aggregation of the forwarding information base: Accounting for locality and churn. *IEEE/ACM Trans. Netw.* 26, 1 (2018).
[19] Roberto Bifulco and Anton Matsiuk. 2015. Towards scalable SDN switches: Enabling faster flow table entries installation. In *SIGCOMM'15*. ACM.
[20] Nicola Bonelli, Stefano Giordano, Gregorio Procissi, and Luca Abeni. 2014. A purely functional approach to packet processing. In *ANCS'14*. ACM.
[21] Michael Borokhovich, Clement Rault, Liron Schiff, and Stefan Schmid. 2018. The show must go on: Fundamental data plane connectivity services for dependable SDNs. *Elsevier Comp. Comm.* 116 (2018).
[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Comput. Commun. Rev.* 44, 3 (2014).
[23] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown et al. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM'13*. ACM.
[24] Gordon Brebner and Weirong Jiang. 2014. High-speed packet processing using reconfigurable computing. *IEEE Micro.* 34, 1 (2014).
[25] Anat Bremler-Barr, David Hay, Idan Moyal, and Liron Schiff. 2017. Load balancing memcached traffic using software defined networking. In *IFIP Networking'17*. IEEE.
[26] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano et al. 2020. hXDP: Efficient software packet processing on FPGA NICs. In *OSDI'20*. USENIX.
[27] Brandon Butler. 2017. What is intent-based networking? Retrieved from https://www.networkworld.com/article/3202699/lan-wan/what-is-intent-based-networking.html.
[28] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: Congestion-based congestion control. *Commun. ACM* 60, 2 (2017).
[29] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. 2007. Ethane: Taking control of the enterprise. In *SIGCOMM'07*. ACM.

[30] Calin Cascaval and Dan Daly. 2017. P4 Architectures. Retrieved from https://p4.org/assets/p4-ws-2017-p4-architectures.pdf.
[31] Huan Chen and Theophilus Benson. 2017. Hermes: Providing tight control over high-performance SDN switches. In *CoNEXT'17*. ACM.
[32] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang et al. 2017. dRMT: Disaggregated programmable switching. In *SIGCOMM'17*. ACM.
[33] P4 Language Consortium. 2020. P4 Expert Roundtable Series 2020. Retrieved from https://p4.org/events/2020-p4-summit/.
[34] P4 Language Consortium. 2020. P4 Runtime. Retrieved from https://p4.org/p4-runtime.
[35] Intel Corporation. 2010. IXP4XX Product Line of Network Processors. Retrieved from https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/ixp4xx-product-line-network-processors-spec-update.pdf.
[36] Intel Corporation. 2013. Intel Ethernet Switch FM6000 Series. Retrieved from https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.
[37] Intel Corporation. 2020. Data Direct I/O. Retrieved from https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.
[38] Intel Corporation. 2020. Intel Tofino. Retrieved from https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.
[39] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. 2012. Camdoop: Exploiting in-network aggregation for big data applications. In *NSDI'12*. USENIX.
[40] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Kőrösi, Balázs Sonkoly, Dávid Haja, Dimitrios P. Pezaros, Stefan Schmid, and Gábor Rétvári. 2019. Tuple space explosion: A denial-of-service attack against a software packet classifier. In *CoNEXT'19*. ACM.
[41] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling flow management for high-performance networks. *ACM SIGCOMM Comput. Commun. Rev.* 41, 4 (2011).
[42] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs et al. 2018. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *NSDI'18*. USENIX.
[43] Huynh Tu Dang, Marco Canini, Fernando Pedone et al. 2016. Paxos made switch-y. *SIGCOMM Comput. Commun. Rev.* 46, 2 (2016).
[44] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. NetPaxos: Consensus at network speed. In *SOSR'15*. ACM.
[45] T. Dargahi, A. Caponi, M. Ambrosin, G. Bianchi, and M. Conti. 2017. A survey on the security of stateful SDN data planes. *IEEE Commun. Surv. Tutor.* 19, 3 (2017).
[46] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior et al. 2012. Large scale distributed deep networks. In *NIPS'12*. Curran Associates Inc.
[47] Richard Draves, Christopher King, Srinivasan Venkatachary, and Brian Zill. 1999. Constructing optimal IP routing tables. In *INFOCOM'99*. IEEE.
[48] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2019. Dataplane equivalence and its applications. In *NSDI'19*. USENIX.
[49] R. Duncan and P. Jungck. 2009. packetC: Language for high performance packet processing. In *HPCC'09*. IEEE.
[50] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. 2008. Towards high performance virtual routers on commodity hardware. In *CoNEXT'08*. ACM.
[51] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2020. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In *ATC'20*. USENIX.
[52] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2013. The road to SDN. *ACM Queue. Queue* 11, 12 (12 2013). DOI: https://doi.org/10.1145/2559899.2560327
[53] Anja Feldman and S. Muthukrishnan. 2000. Tradeoffs for packet classification. In *INFOCOM 2000*. IEEE.
[54] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI'18*. USENIX.
[55] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. 2020. Corundum: An open-source 100-Gbps NIC. In *FPGA'20*. IEEE.
[56] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J. Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich et al. 2013. Languages for software-defined networks. *IEEE Comm. Mag.* 51, 2 (2013).
[57] Open Networking Foundation. 2020. Stratum: Enabling the era of next generation SDN. Retrieved from https://stratumproject.org.

[58] Jing Fu and Jennifer Rexford. 2008. Efficient IP-address lookup with a shared forwarding table for multiple virtual routers. In *CoNEXT'08*. ACM.

[59] Eli Gafni and Dimitri Bertsekas. 1981. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Trans. Commun.* 29, 1 (1981), 11–18.

[60] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling innovation in network function control. *ACM SIGCOMM Comput. Commun. Rev.* 44, 4 (2014).

[61] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data plane performance diagnosis of tcp. In *SOSR'17*. ACM.

[62] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. 2013. Design principles for packet parsers. In *ANCS'13*. ACM.

[63] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing GPU as packet processing accelerator. In *NSDI'17*. USENIX.

[64] Yanlei Gong, Xiong Wang, Mehdi Malboubi, Sheng Wang, Shizhong Xu, and Chen-Nee Chuah. 2015. Towards accurate online traffic matrix estimation in software-defined networks. In *SOSR'15*. ACM.

[65] Andy Gospodarek. 2017. The Rise of SmartNICs—Offloading dataplane traffic to...software. Retrieved from https://youtu.be/AGSy51VlKaM.

[66] Mukul Goyal, Mohd Soperi, Emmanuel Baccelli et al. 2012. Improving convergence speed and scalability in OSPF: A survey. *IEEE Commun. Surv. Tutor.* 14, 2 (2012).

[67] Richard L. Graham, Devendar Bureddy, Pak Lui et al. 2016. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *COM-HPC'16*. IEEE.

[68] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. 2009. Flow processing and the rise of commodity network hardware. *ACM SIGCOMM Comput. Commun. Rev.* 39, 2 (2009).

[69] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *SIGCOMM'18*. ACM.

[70] Jong Hun Han, Prashanth Mundkur, Charalampos Rotsos et al. 2015. Blueswitch: Enabling provably consistent configuration of network switches. In *ACM/IEEE ANCS'15*.

[71] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A software NIC to augment hardware.* Technical Report UCB/EECS-2015-155. UC Berkeley.

[72] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-accelerated software router. In *SIGCOMM'10*. ACM.

[73] F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden. 1970. The interface message processor for the ARPA computer network. In *ACM AFIPS'70 (Spring)*. ACM.

[74] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann et al. 2018. The express data path: Fast programmable packet processing in the operating system kernel. In *CoNEXT'18*. ACM.

[75] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast connectivity recovery entirely in the data plane. In *NSDI'19*. USENIX.

[76] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. 2015. mSwitch: A highly-scalable, modular software switch. In *SOSR'15*. ACM.

[77] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust network measurement for software packet processing. In *SIGCOMM'17*. ACM.

[78] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA workflow for line-rate packet processing. In *FPGA'19*. ACM.

[79] Netronome Systems Inc.2020. Netronome NFP-4000 Flow Processor. Retrieved. from https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf.

[80] Xilinx Inc.2020. Vivado High-Level Synthesis. Retrieved from https://www.xilinx.com/products/design-tools/vivado.html

[81] Raj Jain and Sudipta Paul. 2013. Network virtualization and software defined networking for cloud computing: A survey. *IEEE Commun. Mag.* 51, 11 (2013).

[82] Theo Jepsen, Daniel Alvarez, Nate Foster et al. 2019. Fast string searching on PISA. In *SOSR'19*. ACM.

[83] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng et al. 2014. Millions of little minions: Using packets for low latency network programming and visibility. *ACM SIGCOMM Comput. Commun. Rev.* 44, 4 (2014).

[84] Xin Jin, Eric Keller, and Jennifer Rexford. 2012. Virtual switching without a hypervisor for a more secure cloud. In *HotICE'12*. USENIX.

[85] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-free sub-RTT coordination. In *NSDI'18*. USENIX.

[86] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing key-value stores with fast in-network caching. In *SOSP'17*. ACM.
[87] Chen Jing. 2018. Zero-Copy Optimization for Alibaba Cloud Smart NIC Solution. Retrieved from http://www.alibabacloud.com/blog/zero-copy-optimization-for-alibaba-cloud-smart-nic-solution_593986.
[88] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. 2015. Compiling packet programs to reconfigurable switches. In *NSDI'15*. USENIX.
[89] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI'17*. USENIX.
[90] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. 2015. Raising the bar for using GPUs in software packet processing. In *NSDI'15*. USENIX.
[91] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV service chains at the true speed of the underlying hardware. In *NSDI'18*. USENIX.
[92] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. 2016. CacheFlow: Dependency-aware rule-caching for software-defined networks. In *SOSR'16*. ACM.
[93] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. *SOSR*. In *SOSR'16*. ACM, 10.
[94] S. Keshav and R. Sharma. 1998. Issues and trends in router design. *IEEE Commun. Mag*. 36, 5 (5 1998). DOI: https://doi.org/10.1109/35.668285.
[95] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr. In *NSDI'16*. USENIX.
[96] Changhoon Kim, Anirudh Sivaraman, Naga Katta et al. 2015. In-band network telemetry via programmable data-planes. In *ACM SIGCOMM'15 Demos*. ACM.
[97] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable dynamic network control. In *NSDI'15*. USENIX.
[98] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. 2014. SAX-PAC (Scalable And eXpressive PAcket Classification). In *SIGCOMM'14*. ACM.
[99] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The click modular router. *ACM Trans. Comput. Syst.* 18, 3 (2000).
[100] Teemu Koponen, Keith Amidon, Peter Balland et al. 2014. Network virtualization in multi-tenant datacenters. In *NSDI'14*. USENIX.
[101] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-defined networking: A comprehensive survey. *Proc. IEEE* 103, 1 (2015), 14–76.
[102] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang et al. 2017. NFVnice: Dynamic backpressure and scheduling for NFV service chains. In *SIGCOMM'17*. ACM.
[103] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 8. DOI: http://doi.acm.org/10.1145/359545.359563
[104] Rafael Laufer, Massimo Gallo, Diego Perino, and Anandatirtha Nandugudi. 2016. CliMB: Enabling network function composition with click middleboxes. In *HotMiddlebox'16*. ACM.
[105] Maysam Lavasani, Larry Dennison, and Derek Chiou. 2012. Compiling high throughput network processors. In *ACM/SIGDA FPGA'12*.
[106] P. H. W. Leong. 2008. Recent trends in FPGA architectures and applications. In *DELTA'08*. IEEE.
[107] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gábor Rétvári. 2020. Batchy: Batch-scheduling data flow graphs with service-level objectives. In *USENIX NSDI'20*. USENIX.
[108] Bojie Li, Kun Tan, Layong (Larry) Luo et al. 2016. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *ACM SIGCOMM'16*. ACM.
[109] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. 2016. Be fast, cheap, and in control with SwitchKV. In *NSDI'16*. USENIX.
[110] L. Linguaglossa, S. Lange, S. Pontarelli, G. Rétvári, D. Rossi, T. Zinner, R. Bifulco et al. 2019. Survey of performance acceleration techniques for network function virtualization. *Proc. IEEE* 107, 4 (2019).
[111] Alex X. Liu, Chad R. Meiners, and Eric Torng. 2010. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw.* 18, 2 (Apr. 2010), 490–500. DOI: https://doi.org/10.1109/TNET.2009.2030188
[112] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring connectivity via data plane mechanisms. In *NSDI'13*. USENIX.
[113] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto SmartNICs using iPipe. In *SIGCOMM'19*. ACM.

[114] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward in-network computation with an in-network cache. *SIGOPS Oper. Syst. Rev.* 51, 2 (2017).

[115] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *ATC'19*. USENIX.

[116] Yaoqing Liu, Beichuan Zhang, and Lan Wang. 2013. FIFA: Fast incremental FIB aggregation. In *INFOCOM'13*. IEEE.

[117] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *SIGCOMM'16*. ACM, New York, NY.

[118] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. 2017. Swing state: Consistent updates for stateful and programmable data planes. In *SOSR'17*. ACM.

[119] T. Lévai, G. Pongrácz, P. Megyesi, P. Vörös, S. Laki, F. Németh, and G. Rétvári. 2018. The price for programmability in the software data plane: The vendor perspective. *IEEE J. Select. Areas Commun.* 36, 12 (2018).

[120] Yadi Ma and Suman Banerjee. 2012. A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification. *ACM SIGCOMM Comput. Commun. Rev.* 42, 4 (2012).

[121] Luo Mai, Lukas Rupprecht, Abdul Alim et al. 2014. NetAgg: Using middleboxes for application-specific on-path aggregation in data centres. In *CoNEXT'14*. ACM.

[122] Joao Martins, Mohamed Ahmed, and Costin Raiciu. 2014. ClickOS and the art of network function virtualization. In *NSDI'14*. USENIX.

[123] James McCauley, Aurojit Panda, Arvind Krishnamurthy, and Scott Shenker. 2019. Thoughts on load distribution and the role of programmable switches. *ACM SIGCOMM Comput. Commun. Rev.* 49, 1 (2019).

[124] Nick McKeown. 2017. Programmable forwarding planes are here to stay. In *SIGCOMM NetPL'17*. ACM.

[125] Nick McKeown, Tom Anderson, Hari Balakrishnan et al. 2008. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.* 38, 2 (3 2008).

[126] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *SIGCOMM'17*. ACM.

[127] Oliver Michel, Gábor Rétvári, Roberto Bifulco, and Stefan Schmid. 2020. The Programmable Data Plane Reading List. Retrieved from https://programmabledataplane.review/.

[128] Oliver Michel, John Sonchack, Eric Keller, and Jonathan M. Smith. 2018. Packet-level analytics in software without compromises. In *HotCloud'18*. USENIX.

[129] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy et al. 2016. Universal packet scheduling. In *NSDI'16*. USENIX.

[130] Edgar Costa Molero, Stefano Vissicchio, and Laurent Vanbever. 2018. Hardware-accelerated network control planes. In *HotNets'18*. ACM.

[131] László Molnár, Gergely Pongrácz, Gábor Enyedi et al. 2016. Dataplane specialization for high-performance OpenFlow software switching. In *SIGCOMM'16*. ACM.

[132] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A compiler and run-time system for network programming languages. In *ACM POPL'12*. ACM.

[133] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing software-defined networks. In *NSDI'13*. USENIX.

[134] Masoud Moshref, Apoorv Bhargava, and Adhip Gupta. 2014. Flow-level state transition as a new switch primitive for SDN. In *HotSDN'14*. ACM.

[135] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan et al. 2017. Language-directed hardware design for network performance monitoring. In *SIGCOMM'17*. ACM.

[136] Arista Networks. 2018. Four key trends in the networked use of FPGAs. Retrieved from https://www.arista.com/assets/data/pdf/Whitepapers/Trends-in-FPGA-WP.pdf.

[137] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen et al. 2014. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Commun. Surv. Tutor.* 16, 3 (2014).

[138] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira. 2017. Network slicing for 5G with SDN/NFV: Concepts, architectures, and challenges. *IEEE Commun. Mag.* 55, 5 (2017).

[139] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A framework for NFV applications. In *SOSP'15*. ACM.

[140] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *OSDI'16*. USENIX.

[141] Vincent Douglas Park and M. Scott Corson. 1997. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM'97*. IEEE, 1405–1413.

[142] Justin Pettit, Ben Pfaff, Chris Wright, and Madhu Venugopal. 2015. OVN, Bringing Native Virtual Networking to OVS. Retrieved from http://networkheresy.com/ovn-bringing-native-virtual-networking-to-ovs.

[143] BenPfaff. 2013. RFC 7047: The Open vSwitch Database Management Protocol. Retrieved from https://tools.ietf.org/html/rfc7047.

[144] Ben Pfaff. 2016. Converging approaches in software switches. Retrieved from https://benpfaff.org/ blp/keynote.pdf.
[145] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer et al. 2015. The design and implementation of open vSwitch. In *NSDI'15*. USENIX.
[146] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A programming system for NIC-accelerated network applications. In *OSDI'18*. USENIX.
[147] Gergely Pongrácz, László Molnár, Zoltán Lajos Kis, and Zoltán Turányi. 2013. Cheap silicon: A myth or reality? Picking the right data plane hardware for software defined networking. In *HotSDN'13*. ACM.
[148] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano et al. 2019. FlowBlaze: Stateful packet processing in hardware. In *NSDI'19*. USENIX.
[149] Diana Andreea Popescu, Gianni Antichi, and Andrew W. Moore. 2017. Enabling fast hierarchical heavy hitter detection using programmable data planes. In *SOSR'17*. ACM.
[150] DPDK Project. 2020. Data Plane Development Kit. Retrieved from https://www.dpdk.org/.
[151] The Fast Data Project. 2019. FD.io. Retrieved from https://fd.io.
[152] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. 2013. SIMPLE-fying middlebox policy enforcement using SDN. *ACM SIGCOMM Comput. Commun. Rev.* 43, 4 (2013).
[153] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI'13*. USENIX.
[154] Gábor Rétvári, László Molnár, Gergely Pongrácz, and Gábor Enyedi. 2017. Dynamic compilation and optimization of packet processing programs. In *NetPL'17*. ACM.
[155] Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, and Zalán Heszberger. 2013. Compressing IP forwarding tables: Towards entropy bounds and beyond. In *SIGCOMM'13*. ACM.
[156] Luigi Rizzo. 2012. Netmap: A novel framework for fast packet I/O. In *ATC'12*. USENIX.
[157] Luigi Rizzo and Giuseppe Lettieri. 2012. VALE, a switched ethernet for virtual machines. In *CoNEXT'12*. ACM.
[158] Fabian Ruffy, Tao Wang, and Anirudh Sivarama. 2020. Gauntlet: Finding bugs in compilers for programmable packet processing. In *OSDI'20*. USENIX.
[159] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. 2018. Can the network be the AI accelerator?*SIGCOMM NetCompute*. In *SIGCOMM Workshop on In-Network Computing (NetCompute)*. ACM, 20–25.
[160] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *HotNets'17*. ACM.
[161] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. 2016. In-band synchronization for distributed SDN control planes. *ACM SIGCOMM Comput. Commun. Rev.* 46, 1 (2016).
[162] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer et al. 2000. Smart packets: Applying active networks to network management. *ACM Trans. Comput. Syst.* 18, 1 (2000).
[163] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy et al. 2012. Design and implementation of a consolidated middlebox architecture. In *NSDI'12*. USENIX.
[164] S. Sezer, S. Scott-Hayward, P. K. Chouhan et al. 2013. Are we ready for SDN? Implementation challenges for software-defined networks. *IEEE Comm. Mag.* 51, 7 (2013).
[165] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. PISCES: A programmable, protocol-independent software switch. In *SIGCOMM'16*. ACM.
[166] Muhammad Shahbaz and Nick Feamster. 2015. The case for an intermediate representation for programmable data planes. In *SOSR'15*. ACM.
[167] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson et al. 2017. Evaluating the power of flexible packet processing for network resource allocation. In *NSDI'17*. USENIX.
[168] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating fair queueing on reconfigurable switches. In *NSDI'18*. USENIX.
[169] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *SIGCOMM'19*. ACM.
[170] Zhaogang Shu, Jiafu Wan, Di Li, Jiaxiang Lin, Athanasios V. Vasilakos, and Muhammad Imran. 2016. Security in software-defined networking: Threats and countermeasures. *Springer Mob. Netw. Appl.* 21, 5 (2016).
[171] Giuseppe Siracusano and Roberto Bifulco. 2018. In-network Neural Networks. arxiv:1801.05731.
[172] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu et al. 2016. Packet transactions: High-level programming for line-rate switches. In *SIGCOMM'16*. ACM.
[173] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh et al. 2016. Programmable packet scheduling at line rate. In *SIGCOMM'16*. ACM.
[174] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *SOSR'17*. ACM.
[175] John Sonchack, Oliver Michel, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. 2018. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *ATC'18*. USENIX.

[176] V. Srinivasan, S. Suri, and G. Varghese. 1999. Packet classification using tuple space search. In *SIGCOMM'99*. ACM.
[177] W. P. Stevens, G. J. Myers, and L. L. Constantine. 1974. Structured design. *IBM Syst. J.* 13, 2 (1974).
[178] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling network function parallelism in NFV. In *SIGCOMM'17*. ACM.
[179] Kashyap Thimmaraju, Saad Hermak, Gábor Rétvári, and Stefan Schmid. 2019. MTS: Bringing multi-tenancy to virtual networking. In *ATC'19*. USENIX.
[180] Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, Felicitas Hetzelt, Jan-Pierre Seifert et al. 2017. The vAMP attack: Taking control of cloud systems via the unified packet parser. In *CCSW'17*. ACM.
[181] Kashyap Thimmaraju, Bhargava Shastry, Tobias Fiebig, Felicitas Hetzelt, Jean-Pierre Seifert, Anja Feldmann, and Stefan Schmid. 2018. Taking control of SDN-based cloud systems via the data plane. In *SOSR'18*. ACM.
[182] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in network function virtualization. In *NSDI'18*. USENIX.
[183] Zartash Afzal Uzmi, Markus Nebel, Ahsan Tariq, Sana Jawad, Ruichuan Chen, Aman Shaikh, Jia Wang, and Paul Francis. 2011. SMALTA: Practical and near-optimal FIB aggregation. In *CoNEXT'11*. ACM.
[184] Javier Verdú, Mario Nemirovsky, Jorge García, and Mateo Valero. 2008. Workload characterization of stateful networking applications. In *ISHPC'08*. Springer.
[185] Andreas Voellmy, Junchang Wang, Y. Richard Yang, Bryan Ford, and Paul Hudak. 2013. Maple: Simplifying SDN programming using algorithmic policies. *ACM SIGCOMM Comput. Commun. Rev.* 43, 4 (2013).
[186] Han Wang, Robert Soulé, and Huynh Tu Dang. 2017. P4FPGA: A rapid prototyping framework for P4. In *SOSR'17*.
[187] Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun. 2008. The resource pooling principle. *ACM SIGCOMM Comput. Commun. Rev.* 38, 5 (2008).
[188] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic scaling of stateful network functions. In *NSDI'18*. USENIX.
[189] L. Yang et al. 2004. RFC 3746: Forwarding and Control Element Separation (ForCES) Framework. Retrieved from https://tools.ietf.org/html/rfc3746.
[190] Mao Yang, Yong Li, Depeng Jin, Lieguang Zeng, Xin Wu, and Athanasios Vasilakos. 2015. Software-defined and virtualized future mobile and wireless networks: A survey. *Mob. Netw. Applic.* 20, 1 (2015).
[191] T. Yang, G. Xie, A. X. Liu et al. 2018. Constant IP lookup with FIB explosion. *IEEE/ACM Trans. Netw.* 26, 4 (2018).
[192] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software defined traffic measurement with OpenSketch. In *NSDI'13*.
[193] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. 2015. Scenario-based programming for SDN policies. In *CoNEXT'15*. ACM.
[194] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, high performance ethernet forwarding with CuckooSwitch. In *CoNEXT'13*. ACM.
[195] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro* 34, 5 (2014).
[196] Noa Zilberman, Philip M. Watts, Charalampos Rotsos, and Andrew W. Moore. 2015. Reconfigurable network systems and software-defined networking. *Proc. IEEE* 103, 7 (2015).

Received August 2020; revised January 2021; accepted January 2021