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

Oliver Michel [1], Roberto Bifulco [2], Gábor Rétvári [2], and Stefan Schmid [2]

[1]University of Vienna
[2]Affiliation not available

October 30, 2023

Abstract

Programmable data plane technology enables the systematic reconfiguration of the low-level processing steps applied to network packets and is a key driver in 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 architectures, abstractions, 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.

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
GABOR RETVARI, Budapest University of Technology and Economics (BME), Hungary
STEFAN SCHMID, Faculty of Computer Science, University of Vienna, Austria

Programmable data plane technology enables the systematic reconfiguration of the low-level processing steps applied to network packets and is a key driver in 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 architectures, abstractions, 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.

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 design of network systems, and in particular their main building blocks, i.e., network devices. 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 opposites have ultimately pushed the need (and definition) of 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 programming interface. On the other side, programmability frees device vendors from designing networking 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 operators can leverage to implement custom logic.

This new generation of programmable devices is proving to be especially helpful for operators that now see the advent of large-scale cloud computing, big data applications and massive machine learning, ubiquitous IoT, and the 5G mobile standard. These applications force operators to adopt new ways to architect communication networks, 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

Authors' addresses: Oliver Michel, Faculty of Computer Science, University of Vienna, Vienna, Austria, oliver.michel@univie.ac.at; Roberto Bifulco, NEC Laboratories Europe, Heidelberg, Germany, bifulco@neclab.eu; Gabor Retvari, Budapest University of Technology and Economics (BME), Budapest, Hungary, retvari@tmit.bme.hu; Stefan Schmid, Faculty of Computer Science, University of Vienna, Vienna, Austria, stefan_schmid@univie.ac.at.

functions, on top of the impressive set of features already supported today, including tunneling, load balancing, complex filtering, and enforcing Quality of Service (QoS) constraints.

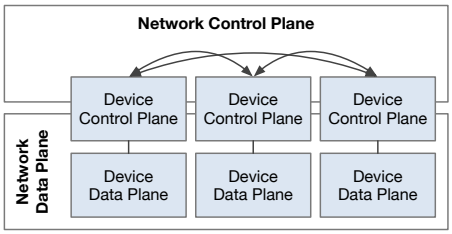
Supporting such an extensive feature set at the required flexibility, dynamicity, performance, and efficiency with traditional fixed function devices requires careful and expensive engineering efforts on the side of device vendors. Such efforts involve the tedious and costly design, manufacturing, testing, and deployment of dedicated hardware components [137, 177], 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 [82, 135, 144, 160, 170]. Leveraging advances in I/O frameworks [87, 169], these programmable software switches can achieve forwarding throughput in 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) [86, 150, 151, 209] and programmable switches [1, 6, 21]. 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 [30] or some dialect of a general purpose language [53, 179].

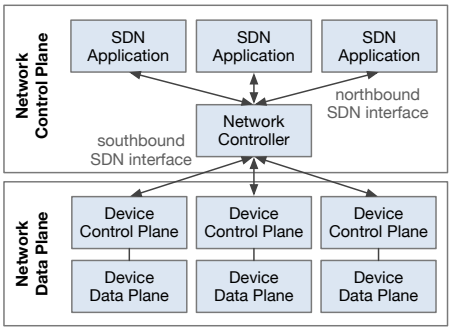
While programmable data plane technologies already gained substantial popularity and adoption, many questions around them remain unanswered. How to adapt and use the elemental packet processing primitives to support the broadest possible selection of network applications at the highest possible performance? How to expose the, potentially very complex, processing logic to the operator for easy, secure, and verifiable configuration? How to abstract, replicate, and monitor ephemeral packet processing state embedded deeply into this logic? Which 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 [105], in this paper *we provide 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 paper to be useful for a broad audience: researchers aiming at getting an overview of the field, students learning about this novel exciting technology, or 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 paper [140]. 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 SDNs as a whole, see [57, 111, 153, 210].

The rest of the paper 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 programmable data plane systems in Sections 4 and 5. In Section 6, we present applications and proposed systems built on top of this technology. Finally, we briefly summarize the work discussed in this paper through a taxonomy in



(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

Section 7, highlight some of the most compelling issues and open problems in the field in Section 8, and conclude in Section 9.

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. As part of this, 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, an edge router, or a gateway), 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 header, and managing the device, including checking its health and performing maintenance operations. The device data plane in turn is responsible solely for executing the packet processing policy set by the device control plane, usually at 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. Through this interaction they create the illusion of a single *network-level control plane* to the rest of the world, executing a virtual global packet forwarding policy in a distributed fashion. Figure 1a shows the network-level and device-level control plane and data plane architecture.

With the introduction of the Software-defined Networking (SDN) paradigm [57, 210], 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 (or *forwarding elements* [3]) do not need to implement all 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 plane. Here, controller-switch communication occurs through a standardized

southbound API, like OpenFlow [138], ForCES [3], the P4Runtime [155], or the Open vSwitch Database Management Protocol [4]. This architecture is depicted in Figure 1b. Note, however, that the device control plane does not fully disappear in the SDN framework; rather, it remains in charge of implementing the control channel towards the remote network control plane and to manage the device data plane (see [43] for a discussion on complete device-level separation of the control plane from the 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* in order to match the packet with the corresponding forwarding policy, which describes the forwarding decision to be applied to the packet (e.g., which output port to use) 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, for instance to increase a flow counter. Once all the modifications are applied, packet headers may be re-generated from packet descriptors (*deparsing*), and finally in the *forwarding* step the packet is sent to an output 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.

These steps can be expected to happen in the reported order; depending on the implementation and underlying device, however, certain processing operations may happen multiple times for a packet. For instance, parsing may examine only the first few bytes of a packet and, only after performing classification and modification/update steps on the parsed data, the remaining bytes may undergo a new parsing step. The classification–modification cycle may also be repeated multiple times, either by sequencing multiple packet processing stages one after the other or by recirculating the packet to the ingress phase of the pipeline for additional processing.

2.3 Data Plane Programmability

With the emergence and adoption of the SDN paradigm over the past several years, 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. As a result, data plane functionality 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. This makes it impossible to, e.g., apply IP routing lookup to packets decapsulated from VXLAN tunnels. Finally, queuing disciplines (e.g., FIFO or priority

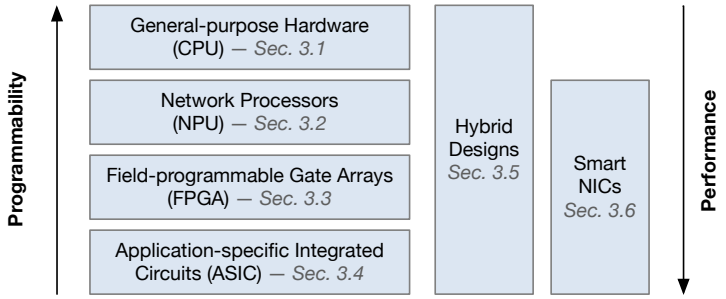


Fig. 2. Overview of hardware architectures programmable data plane systems are commonly built upon.

queuing only, without support for BBR [35]) or 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.

We wish to stress that data plane programmability here 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 [14, 136], 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 to deploy even completely new network protocols in operation; and
- *new applications that can be realized entirely in the data plane leveraging 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 initially was mostly targeted at switches (especially in data center settings), 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 increasingly for general network processing and middlebox functionality (e.g., in firewalls or load balancers) as well [48, 123, 132]. Additionally, programmable network interface cards (often referred to as *SmartNICs*) enable data plane programmability at the edge of the network. These devices can

be realized on top of one of the several different architectures for programmable data planes, or leverage multiple architectures as part of a hybrid design.

In hardware designs, data plane functionality may be implemented in an ASIC (Application-specific Integrated Circuit) [1, 21], an FPGA (Field-programmable Gate Array) [60, 209], or a network processor [6, 86, 151]. These platforms generally offer high performance due to dedicated and specialized hardware components, such as Ternary Content Addressable Memory chips (TCAM) [96] for efficient packet matching. A software data plane device, on the other hand, is one where the data plane executes the entire processing logic on a commodity CPU [45, 56, 77, 82, 144, 157, 160, 178] using fast packet-classification algorithms and data structures [59, 109, 189]. The distinction between hardware and software data planes is somewhat blurred though. For instance, a hardware-based device may still invoke a general-purpose CPU (the "slow path") 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, like Data Direct I/O (DDIO), segmentation offload (TSO/GSO), Receive Side Scaling and Receive Packet Steering (RSS/RPS), and increasingly SmartNIC offloads to run the packet processing logic partially or entirely in hardware. Below we present an overview of the main design points in architectures for programmable data plane systems together with their characteristics, use cases, and trade-offs made. The outline and 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 (NFV) [100, 132, 154] rely on the capability to perform high-performance packet processing with general-purpose servers [106, 157]. Modern virtualized data centers usually have servers running the network access layer [2, 110], using a software switch that connects virtual machines to the physical network [19, 144, 160, 193]. Driven by these requirements, over the past years, software-based packet processing has made significant inroads in the traditionally hardware-dominated network appliance market [54, 73, 162] with several established programmable software switch platforms for efficient network virtualization (VPP [19], BESS [77], FastClick [20], NetBricks [157], PacketShader [78], and ESwitch [144]), user space I/O libraries (PacketShader [8], NetMap [169], Intel DPDK [87], RDMA [98], FD.io [56], and Linux XDP with eBPF [22]), and NFV platforms [101, 112, 192, 196, 207].

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 [123]. 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 [10]. For example, modern multi-processor systems implement Non-uniform Memory Access (NUMA) architectures, which make the relative location of NICs, processors, and memory relevant for the delay and performance of data movements [20, 152]. Optimizing for the system's memory hierarchy can result in performance gains or penalties of several orders of magnitude [19].

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 [169] or Intel DPDK [87], however, cannot use any kernel networking interfaces and need to 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) [80] 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 (DDIO) [55, 85] in order to copy a received packet descriptor directly into the CPU L3 cache bypassing the comparatively slow main memory. Finally, as servers have evolved into multi-processors and multi-core architectures, carefully planning for resources contention cases is important to provide high performance [196].

Given the above hardware properties and constraints, software implementations apply a number of techniques to efficiently use the available resources [10, 123, 160]. Packets are usually processed in batches to amortize the cost of locks on contended resources across the processing pipeline and to improve data locality. Here, locality is important especially for the data required to process a packet, e.g., a lookup table needed for packet classification. Furthermore, it may reduce the amount of misses in the CPU's instruction cache, which may be beneficial for some more complex programs with many instructions [19]. Other typical techniques include adopting data structures that minimize memory usage to better fit in caches [168], aligning data to cache lines to avoid loading multiple cache lines for few additional bytes [144], and distributing packets across different processors keeping flow affinity to avoid cache synchronization issues [101, 192].

Apart from these general optimization techniques, a software implementation can use several further optimization strategies to accelerate packet processing [123]. For instance, ClickOS [135], FastClick [20] and BESS [77] implement a run-to-completion model, in which each packet is entirely processed before processing a second packet on the same core, whereas NFVnice [112] uses standard Linux kernel schedulers and backpressure to control the execution of packet processing functions. Differently, VPP [19] performs 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 [20, 77]. Lazy parsing can be employed to avoid unnecessary and costly parsing operations, e.g., for packets that are to be dropped early [22]. All these different approaches are of course possible due to the flexibility of general-purpose CPUs which do not mandate any specific processing model.

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 (cf. Fig. 3) as an example of a NPU [151].

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. In Netronome terminology, a programmable processing core is named micro-engine (ME). Each ME has 8 threads which share

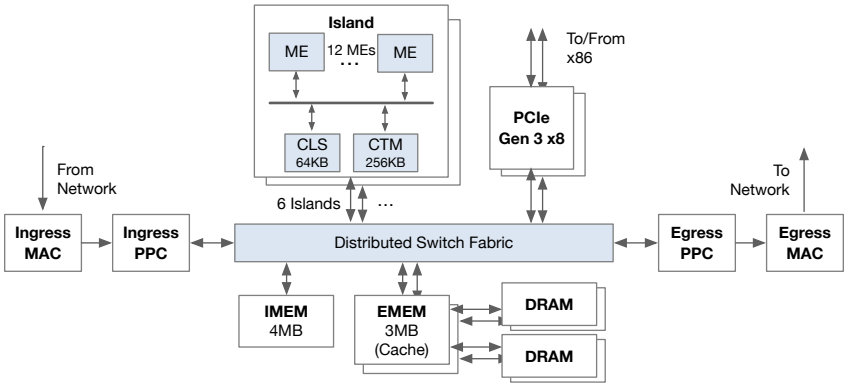


Fig. 3. The architecture of a Netronome NFP's programmable blocks. Some specialized hardware blocks (e.g., for cryptography tasks) are not shown.

local registers that amount for a few KBs of memory. MEs are further organized in islands. Each island has limited shared Static Random Memory Access (SRAM) memory areas of a few hundred KBs: the CLS and CTM memories. Generally, these memory areas are used to host data frequently accessed, and that may be required for the processing of each network packet. Finally, the network processors provide a memory area shared by all islands, the IMEM, of 4MB SRAM, and a memory subsystem that combines two 3MB SRAMs, used as cache, with larger DRAMs, called EMEMs. These larger memories generally host the forwarding tables and access control lists used by the networking subsystem to decide how to forward (or drop) a network packet. All building blocks are interconnected via a high-speed switching fabric, such that MEs can communicate and synchronize with any other ME irrespective of their location. Of course, communications across islands take longer and may impact the performance of a running program. Packets enter and exit the system through arrays of packet processing cores (PPC) that perform packet parsing, classification, and load balancing to the MEs. Media Access Control (MAC) units write and read the packets to and from the network. The Netronome NFP supports different interfaces 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. For instance, it is possible to perform the parsing of just a few bytes in the beginning of the processing, and postpone further parsing only when the need arises. Similarly, the entire packet data is available for processing since data can be stored at the different levels of the processor's memory hierarchy. The handling of the data in relation to the memory hierarchy, however, has a significant impact on the achievable processing speed, and is therefore an important design step (and limiting factor) when programming such devices.

3.3 Field-programmable Gate Arrays

Field-programmable Gate Arrays (FPGA) 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 [119]. High-level synthesis or specialized compilers allow programming FPGAs using languages like C or P4 as opposed to more complex and cumbersome

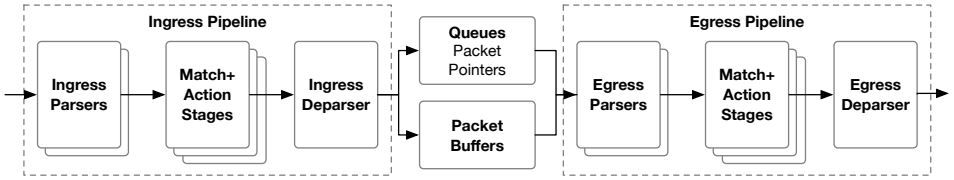


Fig. 4. The architecture of an RMT-like switching ASIC

hardware description languages, such as Verilog [200, 203]. 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 [16, 32, 118]. 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 [60].

The availability and comparatively low cost compared to fully programmable switches (such as devices with a Barefoot Tofino ASIC) make FPGAs particularly interesting for academia to prototype high-performance network data planes. NetFPGA, for example, is a widely available open-source FPGA-accellerated network interface card. The most recent version (FPGA SUME) couples a Xilinx Virtex 7 FPGA with four 10Gb Ethernet ports [209]. A more recent effort in this direction is Corundum [61], which provides an open source platform for implementing a 100Gbps NIC on FPGA. Corundum is a collection of the basic NIC modules and building blocks, which are ready to be implemented on several commercial FPGA cards. FPGAs have also entered the public cloud market with Amazon Web Services offering FPGA-equipped virtual machine instances making the technology even more accessible.

3.4 Application-specific Integrated Circuits

While in the early days of the ARPANET and the Internet, routing and packet processing was performed in software [79], the rapid adoption and increasing scale of the Internet required more efficient hardware-based designs (i.e., ASICs) 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, in order to implement the device's monitoring and control functions, as well as more complex (and uncommon) packet processing functions that the ASIC does not support. The 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. A prominent example of one of the first ASIC-based networking devices is the Juniper M40 router [58] that provided unprecedented 40 Gbit/s routing performance through logically separated control and data plane components within a single chassis together with a highly customized switching chip. 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 hindering flexibility and innovation.

As a result, recently, more flexible and programmable switching chip architectures, such as Reconfigurable Match-action Tables (RMT) [31], the Protocol-independent Switch Architecture (PISA) [37], and implementations, such as Intel Flexpipe [1], Barefoot Tofino [21], or Cavium

Xpliant [6], have been proposed. Programmable data plane devices allow network operators to programmatically change the low-level data plane functionality in order to support novel or custom protocols, to implement custom forwarding or scheduling logic, or to enable new applications that are then entirely executed 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. The tables may further 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 [37]. Figure 4 depicts the RMT reference design for programmable switches. We will further elaborate on the match-action table abstraction used in this design in Section 4.1.2.

3.5 Hybrid Architectures

In addition to the platforms discussed above, interesting hybrid hardware-software designs mixing existing concepts with fresh ideas from distributed systems and multi-processor design have been proposed lately. 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. [162] showed that the overhead of programmability can be relatively low. 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 explored 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 [78]. However, the advantages and disadvantages of this strategy are being actively debated in the systems community [68, 99]. Kalia et al. [99] 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. [68]. 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 [68] 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 [25, 102] architectures are studied which

allow high-speed forwarding even with large rule tables and fast updates, by combining the best of hardware and software processing. In particular, the CacheFlow system [102] 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 trade-offs 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 [188], Sonata [75], and Marple [149]. We further elaborate on these systems in Section 6.1.

3.6 Programmable NICs

Orthogonal to the previously presented architectures, programmable Network Interface Cards, a new platform for programmable data planes, have attracted significant attention in the networking community over the past years. 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. In general, modern NICs implement various features in hardware, such as protocol offloading, multicore support, traffic control, and self-virtualization. In the following, we only focus on the architectural design papers and defer the applications, such as virtualization support, to Section 6.

Without specialization and device-specific optimizations operating systems commonly fail to efficiently leverage and manage the considerable hardware resources provided by modern network interface controllers. To reinvigorate the discussion of the design of NICs, and to overcome current shortcomings, Shinde et al. [182] developed a network stack that represents both the physical capabilities of the network hardware and the current protocol state of the machine as data flow graphs. The implementation of NIC features in hardware can introduce several challenges related to protocol dependencies, limited hardware resources, and incomplete/buggy/non-compliant implementations. The slow evolution of hardware NICs due to increasing design complexity may also not keep up in time with new protocols and rapidly changing network architectures. The SoftNIC architecture [77] has been designed to fill the gap between hardware capabilities and user demands. It implements sophisticated NIC features on a few dedicated processor cores, while assuming only streamlined functionalities in hardware.

A main concern is to simplify the development of server applications that offload computation and data to a NIC accelerator. Floem [161] is a set of programming abstractions for NIC-accelerated applications which 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.

SmartNIC offloading can bring significant performance benefits compared to general-purpose systems by leveraging specialized parallel processors, dedicated subsystems for many networking tasks (e.g., traffic control or encryption), and efficient host communication. Different use cases for offloading distributed applications on SmartNICs are considered in [126]. iPipe is a generic

actor-based offloading framework to run distributed applications on commodity SmartNICs. It is built around a hybrid scheduler that combines first-come-first-serve with deficit round-robin policies to schedule offloading tasks at microsecond-scale precision on SmartNICs.

A fundamental challenge of NICs is related to the noisy neighbor problem. Kumar et al. in [114] systematically characterize how performance isolation can break in virtualization stacks and find a fundamental tradeoff between isolation and efficiency. A new NIC design, PicNIC, the Predictable Virtualized NIC, shares resources efficiently in the common case while rapidly reacting to ensure isolation in order to provide predictable performance for isolated workloads.

Another use case arises in the context of SmartNIC-accelerated servers used to execute microservice-based applications in the data center. 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 [128], 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.

A primary reason for high memory and processing overheads inherent to packet processing applications is the inefficient use of the memory and I/O resources by commodity NICs. FlexNIC [104] implements a new network DMA interface that allows operating systems and applications to install simple packet processing rules into the NIC, which then executes these operations while transferring the packet to the host memory.

4 ABSTRACTIONS

The differences among data plane technologies are often reflected in the packet processing primitives exposed to the control plane and programming 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 for programmable data planes.

4.1 Programmable Packet Processing Pipelines

Flexible packet processing is the core capability of programmable data planes. Today's programmable packet processing pipelines are generally built on top of three fundamental abstractions: the data flow graph abstraction and related switch architectures, the match-action pipeline abstraction, and state machine switch architectures that allow to implement stateful workloads on top of the previous abstractions. We now elaborate on each of these.

4.1.1 Data flow graphs. Early designs for packet processing systems borrowed heavily from generic systems design [190] and machine learning [7], adopting the data-flow graph abstraction to architect programmable switches [147]. 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 the nodes representing elemental computation stages and edges representing the way data moves from one computation stage to the other. A nice 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 with a smooth learning curve.

Perhaps the earliest programmable switch framework adopting the data flow graph abstraction was the Click modular software router [147]. 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 [135], FastClick [20], Vector Packet Processing (VPP) from the FD.io project [56], the Berkeley Extensible Software Switch (BESS, [77]), and NetBricks [157] adopt a similar design, with the difference that the fundamental data unit that moves 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 the built-in Single-Instruction-Multiple-Data (SIMD) instruction sets of modern CPUs results in more efficient software implementations [20, 78, 87]. NetBricks, in addition, introduces a new framework for the isolation of potentially untrusted packet processing nodes, using novel language-level constructs and zero-cost compile-time abstractions [157].

The presence of user-defined functionality abstracted as data flow graph nodes gives a great flexibility and extendibility [117, 135]. 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 [120, 121].

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 [30, 138, 144, 160, 178]. A subset of the packet header fields is used to perform a flow table lookup in the first table 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 [159]. This has the benefit of exposing reconfigurable data plane functionality to operators using the familiar notion of *flows* described by matching *rules* defined over certain header fields (an abstraction extensively used in firewalls and ACLs). Hierachies 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 [138], which in turn borrowed greatly from Ethane [36]. 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 [138]. As illustrated using an example in Figure 5, 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 [160] 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 [144], introducing 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.

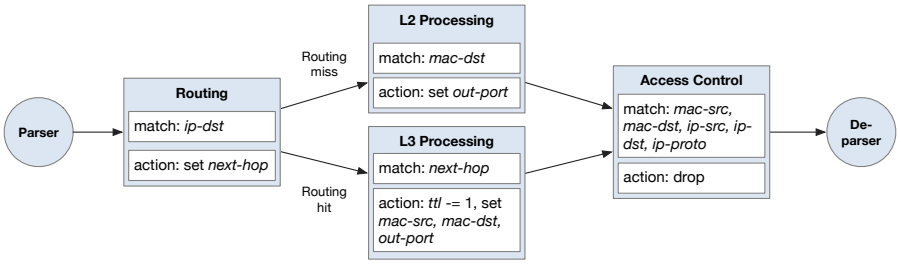


Fig. 5. Simplified match-action table dependency graph for a basic router (inspired by Fig. 3 in [30]).

Driven by the advances in switching ASIC technology, the Reconfigurable Match Tables (RMT) abstraction [31] 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 [39] 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 [30] and the accompanying hardware and software switch projects [1, 21, 178] have been met with increasing enthusiasm from the side of device vendors, operators, and service providers [113, 191].

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 [198]. With the emergence of high-performance packet processing capabilities in software, network functions are increasingly 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 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 [23, 24, 148, 163]. Other more language-focused approaches include Domino [185], 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 [15] 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 [131] 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.

4.2.2 State management in virtualized network functions. NFV promises simplifying middlebox deployment, improving elasticity and fault tolerance while reducing costs. 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 when distributing traffic for dynamic scaling or in the face of failures is non-trivial. There are several lines of work aiming at alleviating this problem. Generally, they can be classified in approaches that (a) keep all state local to a NF and transfer state when required [156, 165, 176], (b) mix local and remote state [65, 166], and (c) use centralized or distributed remote state [97, 202]. Relatable to SwingState [131] in this context is StateAlyzr [106], 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.

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 [67]. 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 when parsing packets. Moreover, headers often only provide partial information (e.g., MPLS) and do not fully specify the subsequent header type, requiring further table lookups or speculative execution.

Implementing low-latency parsers for high-speed networks is particularly challenging. In order 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 may become a security issue, especially for virtual switches [194].

Programmability is another key requirement as header formats may change over time, e.g., due to new standards or due to the desire to support custom headers. Examples of more recent header structures include PBB, VxLAN, NVGRE, STT, or OTV, among many more. In order 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 [67].

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. [186] present a solution which 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 what order to schedule packets and when to schedule them. 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. [142]. 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 good 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 a link, Sharma et al. [181] 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 to transmit 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 (e.g., NICs), doing so compromises on 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 [183] proposes a generalization of the Push-In-First-Out (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 which 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 in what order to schedule amongst the eligible elements. With the hardware design of the PIEO scheduler, also presented in [183], the scalability of this approach is demonstrated.

4.5 Programming Languages and Compilers

An important dimension of programmable data planes regards the programming languages and compilers used to realize the data plane functionality. 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 dataplane 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

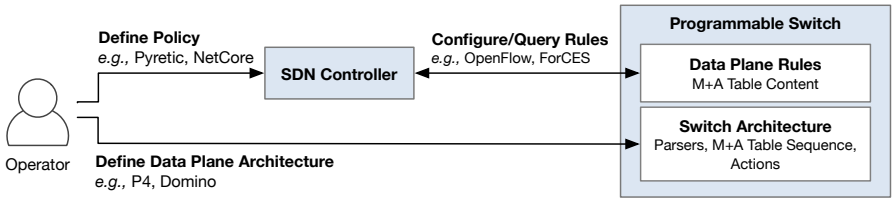


Fig. 6. Comparison of Languages and Protocols used in Programmable Data Planes

differences between these two classes of language abstractions found in programmable data plane systems today are depicted in Figure 6.

4.5.1 SDN policy definition. Languages for SDN programming generally differ in the amount of visibility that should be provided in SDNs (see [44] 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 [62] 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 [145, 146], which provides primitive actions, matching predicates, query policies, and policies. Maple [199] simplifies SDN programming (1) by allowing a programmer to use a standard programming language to design an arbitrary, centralized algorithm, controlling the behavior of the entire network, and (2) 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 [13] 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 [28], 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. Furthermore, challenges arise in the design of compilers and the targeted hardware architectures of such language-based abstractions.

A first and most prominent language abstraction and compiler for specifying low-level packet processing functionality within programmable data planes is P4 [30]. Motivated by the limitation of existing SDN control protocols, such as OpenFlow, which only allow for a fixed set of packet header fields and actions, P4 makes it possible to define hardware packet processing pipelines together with the header parsers and deparsers, match-action tables, and low-level actions that can be applied to each packet. This language abstraction allows for protocol-independent packet processing by matching on arbitrary bit ranges and applying user-defined actions. Abstract P4 programs are then compiled for the underlying data plane target in a separate step. The origins of P4 go back to work by Lavanya et al. [95] who study how to map logical lookup tables to physical tables while meeting data and control dependencies in the program. The authors also present algorithms to generate solutions 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 using a specific control interface, such as the P4 Runtime [74].

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 [200] is a 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 [84], 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. [52] propose a new tool and algorithm, called netdiff, to check the equivalence of related P4 programs and FIB updates in order 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 [17] 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.

5 ALGORITHMS AND HARDWARE REALIZATIONS

The realization of programmable data planes various algorithms, often to be implemented in hardware. In this section, we discuss some of the major algorithms and hardware building blocks used in programmable data planes.

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 [31] 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 which 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 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 [208] have been shown to be 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. Besides optimizing lookup time, improving memory

efficiency of match-action table representations in hardware is an imporant problem given the heavily constrained resources on these devices. 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 [50]) or dynamically (such as FIFA [129], SMALTA [197], or SAIL [204]). Rétvári et al. [168] 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 benefit of this approach.

Inspired by Zipf's law, i.e., 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 be then 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 [25, 26].

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. [63] proposed to use 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 trade-offs between (memory) space and (lookup) time.

Content-addressable memory (CAM) and Ternary CAM (TCAM) chips are the most important component 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 (typically required) multidimensional packet classification [96]. 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. [133] showed how to design a pre-classifier such that a given packet matches at most one entry in the pre-classifier, avoiding rule replication. SAX-PAC in turns exploits the observation that many practical classifiers include lots of independent rules, allowing the corresponding matches to be made in arbitrary order and usually considering only a small subset of dimensions [109]. TCAM Razor [124], furthermore, strives to generate a semantically equivalent packet classifier that requires the least number of TCAM entries. It is also known that the negative space-time tradeoff which seems inherent in the design of classifiers, can sometimes be overcome allowing for, e.g., range constraints [109].

Perhaps the most prominent application of generic wildcard packet classifiers, the Open vSwitch fast-path packet classifier [160] uses a combination of extensive multi-level hierarchical flow-caching and the venerable Tuple Space Search scheme (TSS) [189]. 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 as a hash table that can be searched in constant time. While 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 [41, 42].

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 [38] trades off a nominal amount of TCAM space for assuring improved performance. Also a hybrid software-hardware switch such as ShadowSwitch [27] can help lowering the flow table entry installation time. In particular, since software tables are very fast to be updated, forwarding table updates should happen in software first before being propagated to TCAM to offload software forwarding and to achieve higher overall throughput. Lookups in software should be performed only in case there are no entries matching a packet in hardware. Solutions such as ShadowSwitch further exploit the fact that deleting entries from TCAM is much faster than adding them, hence translating an entry installation to a mix of installation in software tables and deletion from hardware tables.

6 APPLICATIONS

Recently, there is a trend towards moving certain general information-processing functionality formerly implemented either entirely in software on end hosts or on dedicated hardware appliances right 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 at unprecedented rates inside the network. Applications that have been offloaded to the network in this manner include monitoring and telemetry, massive-scale data-processing and 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 occupation levels) or from the energy savings possible by running conventional compute tasks on low-power programmable NICs [127].

One may wonder which types of applications may benefit most from being offloaded into the programmable data plane [172]. 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 (1) *process massive amounts of network-bound data* or have a strong networking component in some way (e.g., implement request-response patterns), (2) *pose stringent latency and/or throughput requirements*, and (3) *can be decomposed into a small set of simple primitives* that lend themselves readily to be implemented partially or entirely on top of the packet processing primitives exposed by the underlying programmable data plane devices.

A typical example would be measurement/telemetry applications, which allow operators to inspect traffic passing through the network at line rate. Today, telemetry mostly occurs “outside the network”, e.g., by mirroring traffic flows to a separate middlebox, raising non-trivial performance and resource consumption concerns. Recently it has been observed that many measurement tasks

can be expressed in terms of simple primitives called *sketches* that can be implemented "inside the network" on programmable switches, resulting in orders of magnitude improvements in speed, latency, and resource consumption at the cost of a minimal loss in precision [83, 187].

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 due to these applications having all the traits that make them most suitable for data plane-based implementations, namely massive traffic scale and stringent performance requirements. Moreover, some applications can be realized using sketches that can efficiently be implemented in switches. The current state-of-the-art involves mirroring monitored traffic to dedicated middleboxes, involving costly duplication of all traffic of interest to an external link; consequently, the efficiency gain with a data plane implementation is enormous. Programmable data planes can be 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. [90] present a solution which not only provides improved visibility to end-hosts but also allows to quickly introduce new data plane functionality, via a new Tiny Packet Program (TTP) interface. Rooted in the work on Smart Packets [174] originally proposed for on-switch network management and monitoring based on the Active Network paradigm [57], TTPs 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 TTPs in-band at line rate, and end hosts perform flexible computation on the network state exposed by the TTPs. The authors also present a number of use case descriptions motivating in-band network telemetry. The general framework for in-band network telemetry (INT) was later presented by Kim et al. in [107].

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 [12]. OpenSketch [205] provides a library of such sketches while UnivMon [130] 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 [83] presented a comprehensive network measurement framework which 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 to 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 [149] identified 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 PFE, but it also requires software offload once the device's SRAM and ALU resources are full. Sonata [75] improved on this hardware restrictive model by more intelligently dividing a query into parts that are executed on the PFE and parts that are executed on a general-purpose software stream processor. Motivated by the limited processing capabilities of software stream processing systems, Sonata introduced a method of iterative refinement, which 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 and also requires relaxing the temporal and logical constraints of a query.

Further applications of in-network measurement are related to heavy hitter detection [164, 187], traffic matrix estimation [69], and TCP performance measurements [66]. First, HashPipe [187] 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. [69] 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 [66] allows to analyze 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 identified 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 [188] 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., 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 a customized, high-performance network analytics platform [141].

6.2 Virtual Switching

Virtual networking is heavily used in data centers and cloud computing infrastructure. At the heart of cloud computing lies the idea 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 [110]. While physically integrated, network virtualization enables logical isolation of resources for each tenant. *Virtual switches* are network components located in the virtualization layer of servers that connect tenants' compute and storage resources (e.g., virtual machines (VMs), storage volumes, etc.), provisioned at the server, to the rest of the data center and the public Internet [2, 88, 110].

Network virtualization and multi-tenancy is typically implemented at the virtual switches that are co-located with the physical servers/hypervisors (e.g, Open vSwitch [160]). 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 [2, 88, 110]. This workspace practically is an overlay network realized through a tunneling protocol, such as VXLAN [5]. As an alternative to this host-based virtual switch model, tagging packets for network virtualization can also be offloaded to the NIC [60, 70].

Despite the widespread deployment of virtual networking [45, 60, 94], providing sufficient (logical and performance) isolation remains a key challenge. For example, serious isolation problems with the Open vSwitch [160] have been reported in [195]: an adversary could not only break out of the VM and attack all applications on the host, but could also manifest as a worm, and compromise an entire data center. Another severe performance isolation vulnerability, also in OVS, was reported in [41, 42] and can result in a low-resource cross-tenant denial-of-service attack. Such attacks may exacerbate concerns surrounding the security and adoption of public clouds [175]. Jin et al. [91] were the first to point out the security weakness of co-locating the virtual switch with the

hypervisor, proposing stronger isolation mechanisms. In response, MTS [193] proposes placing per-tenant virtual switches in virtual machines for increased security isolation.

6.3 In-network Computation

In-network computation typically addresses the performance bottlenecks and scalability limits that massive-scale machine learning and big data frameworks implemented in data centers face [7, 49]. Big data/machine learning applications, such as query processing, graph processing, and deep learning, exhibit a very special communication pattern. First, as in many of these applications the output size is a fraction of the input size, these applications 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, these applications are usually characterized by simple arithmetic/logic operations which make them suitable to parallelization and execution on programmable switches. Third, in many algorithms these operations are also commutative and associative which implies 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 are, however, poorly supported in most network gear incurring significant performance bottlenecks in data center-based deployments.

The first attempt at departing from performing data aggregation at edge servers was Camdoop [40] 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 Camdoop significantly reduces network traffic and provides a performance increase, it requires a custom network topology and is incompatible with common data center infrastructure. Netagg [134] 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 and Apache Solr search. Later, SHArP [72] 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 in data centers.

Liu et al. lay the foundations of a general in-network computation framework by presenting a minimal set of abstractions they call IncBricks [127]: an in-network caching fabric with basic computing primitives based on programmable network devices. The authors in [180] furthermore ask the related general question of how to overcome the limitations imposed by the usually scarce resources provided on programmable switches, like 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 [11] that require explicit support from the network.

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 [171]. More specifically, programmable network devices may be a suitable engine for implementing a CPU's Artificial Neural Networks co-processor. N2Net [184] 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 [89] 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. The authors show that the resulting matching throughput is significantly higher than comparable software implementations.

6.4 Distributed Consensus

Another interesting application for programmable data planes is related to distributed consensus algorithms: the coordination among controllers or switches in order 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, publish-subscribe patterns, and general multi-write key-value stores. Perhaps viewable as a special case of in-network computation, distributed consensus still 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 [92]).

NetPaxos [47] demonstrates the feasibility of implementing the venerable Paxos distributed consensus protocol [115, 116] 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. [46] also show the performance benefits achievable by offloading Paxos into the data plane and describe an implementation in P4.

In-band mechanisms and functionality in the data plane can also be used for synchronization and coordination of other components in distributed systems, such as SDN controllers. Schiff et al. [173] propose a synchronization framework based on atomic transactions implemented in data plane switches and show that their approach allows to realize fundamental consensus primitives in the presence of failures. The authors also discuss applications for consistent policy composition.

Recently, NetChain [92] provides scale-free coordination in data centers 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.

Further interesting applications related to consistency arise in the context of key-value stores. For example, NetCache [93] 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 [122] generalizes this idea by implementing a generic data plane-based key-value query accelerator, with significant improvements in both system throughput and latency. Programmable network switches act as fast key-value caches by keeping track of cached keys and routing requests to the appropriate nodes at line speed based on the query keys encoded in packet headers, so that 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 [188], Marple [149], and IncBricks [127].

Perhaps, an unlikely place to find distributed consensus protocols is in the programmable devices themselves. However, deep inside a typical programmable switch lies a rather complex distributed appliance, with multiple match-action tables, parsers, queues, etc., closely cooperating to perform consistent, high-performance packet processing. It turns out that consistently applying modifications to this pipeline is a rather complex task, in sore need for strong consistency guarantees. Lately, BlueSwitch [76] 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 configuration or the new one, and never an inconsistent mix of the two. This will help avoiding network transients like blackholes and micro-loops that often plague today's operational networks [71].

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 to move functionality for maintaining connectivity under failures into the switches. At the same time, offloading control planes is non-trivial.

The authors in [44] 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, however, can be efficiently offloaded from the controller to the switch itself. In order to meet the needs of high-performance networks, the authors propose and evaluate DevoFlow, a modification of the OpenFlow model which 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. [143] take this idea further and make a general case for offloading control plane protocols (e.g., a routing protocol) 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. Their implementation 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. In order 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 [29]. 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. [125] propose to move 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 like message loss or arbitrary delay from the original algorithm by Gafni and Bertsekas [64] (see also [158]). Holterbach et.al. [81] provide an implementation for automatic data-driven fast reroute entirely in the dataplane. 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 via backup paths without control plane involvement.

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 [9] can also be viewed as a load balancer.

The aim is to implement the "resource pooling" principle using horizontal scaling [201], making a collection of independent resources behave like a single pooled resource in order to exploit statistical multiplexing, load distribution, and improved failure resilience.

A well-known example is HULA [103], a scalable load balancing solution using programmable data planes. HULA is motivated by the shortcomings of Equal-Cost Multi Path (ECMP) as well as of existing congestion-aware load balancing techniques such as CONGA [11]. 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 [139], which leverages programmable switching ASICs to build faster load balancers.

Beyond multipath load balancers, MBalancer [33] 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 the (typically small number of) hot keys in the data plane, duplicating these hot keys to many (or all) Memcached servers, and then adjusting the switches' forwarding tables accordingly.

7 TAXONOMIES FOR PROGRAMMABLE SWITCHES

In Figures 7 and 8, 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 7) and works that leverage programmable data planes in exciting use cases and for novel applications (Figure 8).

Additionally, as an annex to this survey, an annotated reading list for students, practitioners, and researchers interested in the area of programmable data plane technologies is also available online [140].

8 RESEARCH CHALLENGES

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

8.1 Improved Abstractions

Which abstractions provide an optimal tradeoff between supported functionality, resulting performance, and API simplicity?

A first major research challenge revolves around novel *abstractions*. As we have seen, 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 profound enough to allow higher layers to synthesize complex packet processing behavior on top of. Moreover, such an abstraction should be easily exposable to the control plane through a secure and efficient data plane API [138, 155]. It should adequately handle global state embedded in the data plane and provide a well-defined consistency model [202]. It should admit analytic performance models [18, 144] and automatic program transformations for performance optimization [144]. It should separate static semantics from dynamic behavior [167]. And last but not least, it should embrace a convenient mental model that is familiar to network operators and programmers. Not surprisingly, many of the open problems in the field are related to finding the right abstraction for the data plane functionality.

Foundations
- Architectures/Platforms
  - Software: OVS [160], BESS [77], VPP [56], PISCES [178], NetBricks [157]
  - Network Processors: Netronome NFP [151], Intel XScale [86]
  - FPGAs: NetFPGA [209], P4FPGA [200]
  - ASICs: Barefoot Tofino [21], Cavium XPliant [6], Intel Flexpipe [1]
- Abstractions/Building Blocks
  - Match-action: Ethane [36], OpenFlow [138], RMT [31], P4 [30], PISCES [178]
  - Data Flow: Click [147], VPP [56], BESS [77], NetBricks [157]
  - State: FAST [148], OpenState [23], NetBricks [157], Domino [185], SNAP [15], FlowBlaze [163]
- Algorithms
  - Matching: CuckooSwitch [208], FIB Compression [168], Online FIB Aggr. [26]
  - Table Updates: Hermes [38], ShadowSwitch [27]
  - Scheduling: PPS [186], PIEO [183], Approx. Fair Queueing [181], Universal Sched. [142]
- Languages
  - Defining Policy: DevoFlow [44], Pyretic [62], NetCore [145], Maple [199], PFQ [28]
  - Defining Low-level Processing: Packet Programs [95], P4 [30], Domino [185], Netdiff [52]

Fig. 7. Taxonomy of works laying the foundations of programmable data plane technology

Applications
- Monitoring: OpenSketch [205], Marple [149], Sonata [75], INT [107], *Flow [188]
- Switching: OVS [160], OVS Security [195], AccelNet [60], Network Virt. [110]
- In-network computation: Camdoop [40], IncBricks [127], NetAI [171], N2Net [184]
- Consensus: NetPaxos [47], Switchy Paxos [46], NetChain [92], NetCache [93]
- Resiliency: Connectivity in the Data Plane [125], Blink [81], Hedera [9]
- Load balancing: CONGA [11], SilkRoad [139], Hula [103], MBalancer [33]

Fig. 8. Taxonomy of key applications built on top of programmable data planes

8.2 Efficient Reconfigurability

How to support more efficient yet consistent reconfigurability in the data plane?

A related issue regards the support for reconfigurability. Alongside the move from the rigid programming model of OpenFlow to the more flexible P4 world, comes 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, but extends to further critical packet processing operations, and the reconfigurability thereof, ranging from programmable packet parsing [67] to universal scheduling and queuing schemes [142, 186]. In particular, changing data plane behavior at runtime without disrupting packet processing [188] remains an open problem.

8.3 Scalability

How to realize high performance implementations of data planes, especially stateful ones?

The need to scale systems to handle massive workloads increasingly pushes designers to explore more complex solutions that handle some state already in the data plane [93, 139, 180]. 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 [31, 51].

8.4 Network Automation

How to design more automated and self-adjusting networks that are able to map high-level policies to the underlying physical infrastructure and autonomously adapt their configuration and operation to changing demands or failures?

A major current trend in networking concerns *automation*. Over the last years, the vision of "self-driving" communication networks which 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 [34]. 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 [62, 95, 108, 146, 199, 206]. Yet, it is still not clear how to best expose data plane functionality to the operator offering the maximum programming freedom while masking the underlying complexities efficiently. Ideally, an "intent-based data plane compiler" should actively attempt to find the data plane representation that would yield the highest performance [144] with the minimal data plane footprint [124, 168], built on a firm theoretical foundation for optimizing data plane programs and reasoning about performance [18, 144].

8.5 Verification and Monitoring

How to design efficient verification and monitoring frameworks which allow the operator or intent layer to (provably) test the correctness and performance of the network state and load?

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 sufficiency 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 in mind [108], which may require the definition of new abstractions. In general, given the mission-critical role the data plane plays, the success of novel data plane technologies will depend on the reliability guarantees they can provide.

These and other research directions, related to abstractions, performance, automation, and security, will likely continue to require the attention of researchers for many years in an effort to find improved tradeoffs and new opportunities in an evolving context.

9 CONCLUSION

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 hence also support niche solutions: solutions which would not have been worthwhile realizing for vendors, due to the small-scale market. While the body of existing work in this field covered in this survey is already vast, the field is still rapidly evolving and we believe network programmability is still in its infancy.

REFERENCES

[1] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switchfm6000-series-brief.pdf.

[2] Ovn, bringing native virtual networking to ovs. http://networkheresy.com/ovn-bringing-native-virtual-networking-to-ovs/.

[3] Rfc 3746: Forwarding and control element separation (forces) framework. https://tools.ietf.org/html/rfc3746.

[4] Rfc 7047: The open vswitch database management protocol. https://tools.ietf.org/html/rfc3746.

[5] Rfc 7348: Virtual extensible local area network (VXLAN). https://tools.ietf.org/html/rfc7348.

[6] XPliant ethernet switch product family. http://www.cavium.com/XPliant-Ethernet-Switch-ProductFamily.html.

[7] Abadi, M., Barham, P., Chen, J., et al. TensorFlow: a system for large-scale machine learning. In *Proc. USENIX OSDI '16* (2016), USENIX.

[8] Advanced Networking Lab/KAIST. Packet I/O Engine. https://github.com/PacketShader/Packet-IO-Engine.

[9] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *Proc. USENIX NSDI '10* (2010).

[10] Alipourfard, O., and Yu, M. Decoupling algorithms and optimizations in network functions. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (2018), HotNets '18, p. 71–77.

[11] Alizadeh, M., Edsall, T., Dharmapurikar, S., et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proc. ACM SIGCOMM '14* (2014).

[12] Alon, N., Matias, Y., and Szegedy, M. The space complexity of approximating the frequency moments. In *Proc. ACM STOC '96* (1996).

[13] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. Netkat: Semantic foundations for networks. In *Proc. ACM POPL '14* (2014).

[14] Antichi, G., Benson, T., Foster, N., Ramos, F. M. V., and Sherry, J. Programmable Network Data Planes (Dagstuhl Seminar 19141). *Dagstuhl Reports 9*, 3 (2019), 178–201.

[15] Arashloo, M. T., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. SNAP: Stateful network-wide abstractions for packet processing. In *Proc. ACM SIGCOMM '16* (2016).

[16] Arista. Whitepaper: Four key trends in the networked use of fpgas. https://www.arista.com/assets/data/pdf/Whitepapers/Trends-in-FPGA-WP.pdf.

[17] Bai, J., Bi, J., Kuang, P., et al. NS4: Enabling programmable data plane simulation. In *Proc. ACM SOSR '18*.

[18] Bansal, M., Schulman, A., and Katti, S. Atomix: A framework for deploying signal processing applications on wireless infrastructure. In *Proc. USENIX NSDI '15* (2015), USENIX.

[19] Barach, D., Linguaglossa, L., Marion, D., Pfister, P., Pontarelli, S., and Rossi, D. High-speed software data plane via vectorized packet processing. *IEEE Comm. 56*, 12 (2018).

[20] Barbette, T., Soldani, C., and Mathy, L. Fast userspace packet processing. In *Proc. ANCS '15* (2015), IEEE.

[21] Barefoot Networks. Barefoot Tofino: world's fastest P4-programmable Ethernet switch ASICs. https://barefootnetworks.com/products/brief-tofino/.

[22] Bertin, G. XDP in practice: Integrating XDP into our DDoS mitigation pipeline. In *Netdev* (2017).

[23] Bianchi, G., Bonola, M., Capone, A., and Cascone, C. OpenState: programming platform-independent stateful Openflow applications inside the switch. *SIGCOMM CCR 44*, 2 (4 2014).

[24] Bianchi, G., Bonola, M., Pontarelli, S., et al. Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing. *CoRR abs/1605.01977* (2016).

[25] Bienkowski, M., Marcinkowski, J., Pacut, M., Schmid, S., and Spyra, A. Online tree caching. In *Proc. SPAA '17*.

[26] Bienkowski, M., Sarrar, N., Schmid, S., and Uhlig, S. Online aggregation of the forwarding information base: Accounting for locality and churn. *IEEE/ACM TON 26*, 1 (2018).

[27] Bifulco, R., and Matsiuk, A. Towards scalable SDN switches: Enabling faster flow table entries installation. In *Proc. ACM SIGCOMM '15* (2015), ACM.

[28] Bonelli, N., Giordano, S., Procissi, G., and Abeni, L. A purely functional approach to packet processing. In *Proc. ANCS '14* (2014), ACM.

[29] Borokhovich, M., Rault, C., Schiff, L., and Schmid, S. The show must go on: Fundamental data plane connectivity services for dependable sdns. *Elsevier Comp. Comm. 116* (2018).

[30] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR 44*, 3 (2014).

[31] Bosshart, P., Gibb, G., Kim, H.-S., et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proc. ACM SIGCOMM '13* (2013), ACM.

[32] Brebner, G., and Jiang, W. High-speed packet processing using reconfigurable computing. *IEEE Micro 34*, 1.

[33] Bremler-Barr, A., Hay, D., Moyal, I., and Schiff, L. Load balancing memcached traffic using software defined networking. In *IFIP Networking* (2017), pp. 1–9.

[34] Butler, B. What is intent-based networking? https://www.networkworld.com/article/3202699/lan-wan/what-is-intent-based-networking.html, 2017.

[35] Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., and Jacobson, V. Bbr: Congestion-based congestion control. *Commun. ACM 60*, 2 (January 2017), 58–66.

[36] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: Taking control of the enterprise. In *Proc. ACM SIGCOMM '07* (2007), ACM.

[37] Cascaval, C., and Daly, D. P4 architectures. https://p4.org/assets/p4-ws-2017-p4-architectures.pdf.

[38] Chen, H., and Benson, T. Hermes: providing tight control over high-performance SDN switches. In *Proc. CoNEXT '17* (2017), ACM.

[39] Chole, S., Fingerhut, A., Ma, S., et al. dRMT: disaggregated programmable switching. In *Proc. ACM SIGCOMM '17*.

[40] Costa, P., Donnelly, A., Rowstron, A., and O'Shea, G. Camdoop: Exploiting in-network aggregation for big data applications. In *Proc. USENIX NSDI '12* (2012), USENIX.

[41] Csikor, L., Divakaran, D. M., Kang, M. S., Kőrösi, A., Sonkoly, B., Haja, D., Pezaros, D. P., Schmid, S., and Rétvári, G. Tuple space explosion: A denial-of-service attack against a software packet classifier. In *Proc. ACM CoNEXT '19* (2019).

[42] Csikor, L., Rothenberg, C., Pezaros, D. P., Schmid, S., Toka, L., and Retvari, G. Policy injection: A cloud dataplane dos attack. In *Proc. ACM SIGCOMM Posters and Demos* (2018).

[43] Csikor, L., Toka, L., Szalay, M., Pongrácz, G., Pezaros, D. P., and Rétvári, G. Harmless: Cost-effective transitioning to sdn for small enterprises. In *Proceedings of IFIP Netwoking* (2018).

[44] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. Devoflow: scaling flow management for high-performance networks. In *ACM SIGCOMM CCR* (2011), vol. 41, ACM.

[45] Dalton, M., Schultz, D., Adriaens, J., et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. USENIX NSDI '18* (2018), USENIX.

[46] Dang, H. T., Canini, M., Pedone, F., and Soulé, R. Paxos made switch-y. *ACM SIGCOMM CCR 46*, 2 (2016).

[47] Dang, H. T., Sciascia, D., Canini, M., et al. NetPaxos: Consensus at network speed. In *Proc. ACM SOSR '15*.

[48] Dargahi, T., Caponi, A., Ambrosin, M., Bianchi, G., and Conti, M. A survey on the security of stateful SDN data planes. *IEEE Communications Surveys Tutorials 19*, 3 (thirdquarter 2017), 1701–1725.

[49] Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*.

[50] Draves, R., King, C., Venkatachary, S., et al. Constructing optimal ip routing tables. In *Proc. IEEE INFOCOM '99*.

[51] Drepper, U. What every programmer should know about memory. Web page, last accessed April 21 2018, 2007.

[52] Dumitrescu, D., Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. Dataplane equivalence and its applications. In *Proc. USENIX NSDI '19* (2019).

[53] Duncan, R., and Jungck, P. packetC: language for high performance packet processing. In *2009 11th IEEE International Conference on High Performance Computing and Communications* (June 2009), IEEE HPCC 2009, pp. 450–457.

[54] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., and Mathy, L. Towards high performance virtual routers on commodity hardware. In *Proc. CoNEXT '08* (2008), ACM.

[55] Farshin, A., Roozbeh, A., Jr., G. Q. M., and Kostić, D. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In *Proc. USENIX ATC '20* (2020), USENIX.

[56] FD.io. The fast data project. project website, 2016.

[57] Feamster, N., Rexford, J., and Zegura, E. The road to SDN. *Queue 11*, 12 (12 2013), 20:20–20:40.

[58] Fedorkow, G. The juniper m40 router. https://computerhistory.org/blog/the-juniper-m40-router/.

[59] Feldman, A., and Muthukrishnan, S. Tradeoffs for packet classification. In *Proc. INFOCOM 2000* (2000).

[60] Firestone, D., Putnam, A., Mundkur, S., et al. Azure accelerated networking: Smartnics in the public cloud. In *Proc. USENIX NSDI '18* (2018), USENIX.

[61] Forencich, A., Snoeren, A. C., Porter, G., and Papen, G. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines* (2020).

[62] Foster, N., Guha, A., Reitblatt, M., Story, A., Freedman, M. J., Katta, N. P., Monsanto, C., Reich, J., Rexford, J., Schlesinger, C., et al. Languages for software-defined networks. *IEEE Comm. Magazine 51*, 2 (2013).

[63] Fu, J., and Rexford, J. Efficient ip-address lookup with a shared forwarding table for multiple virtual routers. In *Proceedings of the 2008 ACM CoNEXT Conference* (New York, NY, USA, 2008), ACM CoNEXT '08, ACM, pp. 21:1–21:12.

[64] Gafni, E., and Bertsekas, D. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE transactions on communications 29*, 1 (1981), 11–18.

[65] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. Opennf: Enabling innovation in network function control. *SIGCOMM Comput. Commun. Rev. 44*, 4 (August 2014), 163–174.

[66] Ghasemi, M., Benson, T., and Rexford, J. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research* (2017), ACM, pp. 61–74.

[67] Gibb, G., Varghese, G., Horowitz, M., and McKeown, N. Design principles for packet parsers. In *Proc. ACM/IEEE ANCS '13* (2013).

[68] Go, Y., Jamshed, M. A., Moon, Y., Hwang, C., and Park, K. APUNet: revitalizing GPU as packet processing accelerator. In *Proc. USENIX NSDI '17* (2017), USENIX.

[69] Gong, Y., Wang, X., Malboubi, M., Wang, S., Xu, S., and Chuah, C.-N. Towards accurate online traffic matrix estimation in software-defined networks. In *Proc. ACM SOSR '15* (2015), ACM.

[70] Gospodarek, A. The Rise of SmartNICs – offloading dataplane traffic to...software. https://youtu.be/AGSy51VlKaM, 2017. Open vSwitch Conference.

[71] Goyal, M., Soperi, M., Baccelli, E., Choudhury, G., Shaikh, A., Hosseini, H., and Trivedi, K. Improving convergence speed and scalability in ospf: A survey. *IEEE Communications Surveys & Tutorials 14*, 2 (2012), 443–463.

[72] Graham, R. L., Bureddy, D., Lui, P., et al. Scalable hierarchical aggregation protocol (sharp): a hardware architecture for efficient data reduction. In *Proc. IEEE COM-HPC '16* (2016), IEEE.

[73] Greenhalgh, A., Huici, F., Hoerdt, M., Papadimitriou, P., Handley, M., and Mathy, L. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev. 39*, 2 (March 2009), 20–26.

[74] Group, P. A. W. P4runtime specification. https://github.com/p4lang/p4runtime.

[75] Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., and Willinger, W. Sonata: Query-driven Streaming Network Telemetry. In *Proc. ACM SIGCOMM '18* (2018), ACM.

[76] Han, J. H., Mundkur, P., Rotsos, C., Antichi, G., Dave, N., Moore, A. W., and Neumann, P. G. Blueswitch: enabling provably consistent configuration of network switches. In *Proc. ACM/IEEE ANCS '15* (2015), IEEE.

[77] Han, S., Jang, K., Panda, A., Palkar, S., Han, D., and Ratnasamy, S. SoftNIC: A software NIC to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[78] Han, S., Jang, K., Park, K., and Moon, S. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New York, NY, USA, 2010), ACM SIGCOMM '10, ACM, pp. 195–206.

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

[80] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., et al. The express data path: Fast programmable packet processing in the operating system kernel. In *Proc. ACM CoNEXT '18* (2018), ACM.

[81] Holterbach, T., Molero, E. C., Apostolaki, M., Dainotti, A., Vissicchio, S., and Vanbever, L. Blink: Fast connectivity recovery entirely in the data plane. In *Proc. USENIX NSDI '19* (2019), USENIX.

[82] Honda, M., Huici, F., Lettieri, G., and Rizzo, L. mSwitch: a highly-scalable, modular software switch. In *Proc. ACM SOSR '15* (2015), ACM.

[83] Huang, Q., Jin, X., Lee, P. P. C., Li, R., Tang, L., Chen, Y.-C., and Zhang, G. Sketchvisor: Robust network measurement for software packet processing. In *Proc. ACM SIGCOMM '17* (2017), ACM.

[84] Ibanez, S., Brebner, G., McKeown, N., and Zilberman, N. The p4->netfpga workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY, USA, 2019), FPGA '19, Association for Computing Machinery, p. 1–9.

[85] Intel. Intel data direct i/o. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[86] Intel. IXP4XX product line of network processors. http://www.intel.com/content/www/us/en/intelligent-systems/previous-generation/intel-ixp4xx-intel-network-processor-product-line.html.

[87] Intel. Intel DPDK: Data Plane Development Kit. http://dpdk.org, 2016.

[88] Jain, R., and Paul, S. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communication Magazine 51*, 11 (2013).

[89] Jepsen, T., Alvarez, D., Foster, N., et al. Fast string searching on pisa. In *Proc. ACM SOSR '19*.

[90] Jeyakumar, V., Alizadeh, M., Geng, Y., Kim, C., and Mazières, D. Millions of little minions: Using packets for low latency network programming and visibility. In *ACM SIGCOMM CCR* (2014), vol. 44, ACM.

[91] Jin, X., Keller, E., and Rexford, J. Virtual switching without a hypervisor for a more secure cloud. In *Proc. USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (HotICE)* (2012).

[92] Jin, X., Li, X., Zhang, H., Foster, N., Lee, J., Soulé, R., Kim, C., and Stoica, I. Netchain: Scale-free sub-rtt coordination. In *Proc. USENIX NSDI '18* (2018), USENIX.

[93] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. NetCache: balancing key-value stores with fast in-network caching. In *Proc. ACM SOSP '17* (2017), ACM.

[94] Jing, C. Zero-Copy Optimization for Alibaba Cloud Smart NIC Solution. http://www.alibabacloud.com/blog/zero-copy-optimization-for-alibaba-cloud-smart-nic-solution_593986, 2018. Accessed: 03-01-2019.

[95] Jose, L., Yan, L., Varghese, G., and McKeown, N. Compiling packet programs to reconfigurable switches. In *Proc. USENIX NSDI '15* (2015), USENIX.

[96] K., P., and S., A. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits 41*, 3 (2006), 712–727.

[97] Kablan, M., Alsudais, A., Keller, E., and Le, F. Stateless network functions: Breaking the tight coupling of state and processing. In *Proc. USENIX NSDI '17* (2017).

[98] Kalia, A., Kaminsky, M., and Andersen, D. G. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM '14* (2014), ACM.

[99] Kalia, A., Zhou, D., Kaminsky, M., and Andersen, D. G. Raising the bar for using GPUs in software packet processing. In *Proc. USENIX NSDI '15* (2015), USENIX.

[100] Katia Obraczka, Christian Rothenberg, A. R. SDN, NFV and their role in 5G, 2016. ACM SIGCOMM Tutorial.

[101] Katsikas, G. P., Barbette, T., Kostić, D., Steinert, R., and Jr., G. Q. M. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX NSDI* (2018), pp. 171–186.

[102] Katta, N., Alipourfard, O., Rexford, J., and Walker, D. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. ACM SOSR '16* (2016), ACM.

[103] Katta, N., Hira, M., Kim, C., Sivaraman, A., and Rexford, J. Hula: Scalable load balancing using programmable data planes. In *ACM Symposium on SDN Research (SOSR)* (2016), ACM, p. 10.

[104] Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High performance packet processing with FlexNIC. *SIGPLAN Not. 51*, 4 (March 2016), 67–81.

[105] Keshav, S., and Sharma, R. Issues and trends in router design. *IEEE Comm. Magazine 36*, 5 (5 1998).

[106] Khalid, J., Gember-Jacobson, A., Michael, R., Abhashkumar, A., and Akella, A. Paving the way for nfv: Simplifying middlebox modifications using StateAlyzr. In *Proc. USENIX NSDI '16* (2016), USENIX.

[107] Kim, C., Sivaraman, A., Katta, N., et al. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM '15 Demos* (2015).

[108] Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., and Clark, R. Kinetic: Verifiable dynamic network control. In *Proc. USENIX NSDI '15* (2015), USENIX.

[109] Kogan, K., Nikolenko, S., Rottenstreich, O., Culhane, W., and Eugster, P. SAX-PAC (Scalable And eXpressive PAcket Classification). In *Proc. ACM SIGCOMM '14* (2014), ACM.

[110] Koponen, T., Amidon, K., Balland, P., et al. Network virtualization in multi-tenant datacenters. In *Proc. USENIX NSDI '14* (2014), USENIX.

[111] Kreutz, D., Ramos, F. M., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE 103*, 1 (2015), 14–76.

[112] Kulkarni, S. G., Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K. K., Wood, T., Arumaithurai, M., and Fu, X. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *ACM SIGCOMM* (2017), pp. 71–84.

[113] Kumar, N. Juniper advancing disaggregation through P4 runtime integration, 2018. https://forums.juniper.net/t5/The-New-Network/Juniper-Advancing-Disaggregation-Through-P4-Runtime-Integration/ba-p/319195.

[114] Kumar, P., Dukkipati, N., Lewis, N., et al. PicNIC: Predictable virtualized NIC. In *Proc. ACM SIGCOMM '19*.

[115] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978).

[116] Lamport, L. Fast Paxos. *Distributed Computing 19*, 2 (2006), 79–103.

[117] Laufer, R., Gallo, M., Perino, D., and Nandugudi, A. CliMB: enabling network function composition with Click middleboxes. In *Proc. ACM HotMiddlebox '16* (2016), ACM.

[118] Lavasani, M., Dennison, L., and Chiou, D. Compiling high throughput network processors. In *Proc. ACM/SIGDA FPGA '12* (2012), ACM.

[119] Leong, P. H. W. Recent trends in fpga architectures and applications. In *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)* (2008), pp. 137–141.

[120] Lévai, T., Németh, F., Raghavan, B., and Rétvári, G. Batchy: Batch-scheduling data flow graphs with service-level objectives. In *Proc. USENIX NSDI '20* (2020), USENIX.

[121] Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proc. ACM SIGCOMM '16* (2016), ACM.

[122] Li, X., Sethi, R., Kaminsky, M., Andersen, D. G., and Freedman, M. J. Be fast, cheap and in control with switchkv. In *Proc. USENIX NSDI '16* (2016), USENIX.

[123] Linguaglossa, L., Lange, S., Pontarelli, S., Rétvári, G., Rossi, D., Zinner, T., Bifulco, R., et al. Survey of performance acceleration techniques for network function virtualization. *Proceedings of the IEEE* (2019), 1–19.

[124] Liu, A. X., Meiners, C. R., and Torng, E. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw. 18*, 2 (April 2010), 490–500.

[125] Liu, J., Panda, A., Singla, A., Godfrey, B., Schapira, M., and Shenker, S. Ensuring connectivity via data plane mechanisms. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013), pp. 113–126.

[126] Liu, M., Cui, T., Schuh, H., Krishnamurthy, A., Peter, S., and Gupta, K. Offloading distributed applications onto SmartNICs using iPipe. In *Proc. ACM SIGCOMM '19* (2019), ACM.

[127] Liu, M., Luo, L., Nelson, J., Ceze, L., Krishnamurthy, A., and Atreya, K. Incbricks: Toward in-network computation with an in-network cache. *SIGOPS Oper. Syst. Rev. 51*, 2 (April 2017), 795–809.

[128] Liu, M., Peter, S., Krishnamurthy, A., and Phothilimthana, P. M. E3: Energy-efficient microservices on smartnic-accelerated servers. In *Proc. ATC '19* (2019), USENIX.

[129] Liu, Y., Zhang, B., and Wang, L. Fifa: Fast incremental fib aggregation. In *Proc. IEEE INFOCOM '13* (2013), IEEE.

[130] Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., and Braverman, V. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proc. ACM SIGCOMM '16* (New York, NY, USA, 2016), ACM.

[131] Luo, S., Yu, H., and Vanbever, L. Swing state: Consistent updates for stateful and programmable data planes. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2017), ACM SOSR '17, ACM, pp. 115–121.

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

[133] Ma, Y., and Banerjee, S. A smart pre-classifier to reduce power consumption of tcams for multi-dimensional packet classification. *ACM SIGCOMM Comput. Commun. Rev. 42*, 4 (August 2012), 335–346.

[134] Mai, L., Rupprecht, L., Alim, A., Costa, P., Migliavacca, M., Pietzuch, P., and Wolf, A. L. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *Proc. ACM CoNEXT '14*, ACM.

[135] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. ClickOS and the art of network function virtualization. In *Proc. NSDI '14* (2014), USENIX.

[136] McCauley, J., Panda, A., Krishnamurthy, A., and Shenker, S. Thoughts on load distribution and the role of programmable switches. *SIGCOMM Comput. Commun. Rev. 49*, 1 (February 2019), 18–23.

[137] McKeown, N. Programmable forwarding planes are here to stay. In *Proc. ACM SIGCOMM NetPL '17* (2017).

[138] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM CCR 38*, 2 (3 2008), 69–74.

[139] Miao, R., Zeng, H., Kim, C., Lee, J., and Yu, M. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. ACM SIGCOMM '17* (2017), ACM.

[140] Michel, O., Rétvári, G., Bifulco, R., and Schmid, S. The programmable data plane reading list. https://programmabledataplane.review/.

[141] Michel, O., Sonchack, J., Keller, E., and Smith, J. M. Packet-level analytics in software without compromises. In *Proc. USENIX HotCloud '18* (2018), USENIX.

[142] Mittal, R., Agarwal, R., Ratnasamy, S., and Shenker, S. Universal packet scheduling. In *Proc. USENIX NSDI '16*.

[143] Molero, E. C., Vissicchio, S., and Vanbever, L. Hardware-accelerated network control planes. In *Proc. ACM HotNets '18* (2018), ACM.

[144] Molnár, L., Pongrácz, G., Enyedi, G., Kis, Z. L., Csikor, L., Juhász, F., Kőrösi, A., and Rétvári, G. Dataplane specialization for high-performance OpenFlow software switching. In *Proc. ACM SIGCOMM '16* (2016), ACM.

[145] Monsanto, C., Foster, N., Harrison, R., and Walker, D. A compiler and run-time system for network programming languages. In *Proc. ACM POPL '12* (2012), ACM.

[146] Monsanto, C., Reich, J., Foster, N., Rexford, J., and Walker, D. Composing software-defined networks. In *Proc. USENIX NSDI '13* (2013), USENIX.

[147] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F. The Click modular router. In *ACM Trans. on Computer Systems* (2000), ACM Trans. on Computer Systems 2000.

[148] Moshref, M., Bhargava, A., Gupta, A., Yu, M., and Govindan, R. Flow-level state transition as a new switch primitive for SDN. In *Proc. ACM HotSDN '14*.

[149] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed hardware design for network performance monitoring. In *Proc. ACM SIGCOMM '17* (2017), ACM.

[150] NetCope. FPGA NICs Specification. https://www.netcope.com/en/products/fpga-boards.

[151] Netronome. Netronome NFP-6000 Flow Processor. https://www.netronome.com/m/documents/PB_NFP-6000_.pdf.

[152] Neugebauer, R., Antichi, G., Zazo, J. F., Audzevich, Y., López-Buedo, S., and Moore, A. W. Understanding pcie performance for end host networking. In *Proc. ACM SIGCOMM '18*.

[153] Nunes, B. A. A., Mendonca, M., Nguyen, X.-N., Obraczka, K., and Turletti, T. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Comm. Surveys & Tutorials 16*, 3 (2014).

[154] Ordonez-Lucena, J., et al. Network slicing for 5G with SDN/NFV: Concepts, architectures, and challenges. *IEEE Communications Magazine 55*, 5 (2017), 80–87.

[155] P4.org. P4 Runtime. https://p4.org/p4-runtime.
[156] Palkar, S., Lan, C., Han, S., Jang, K., Panda, A., Ratnasamy, S., Rizzo, L., and Shenker, S. E2: A framework for nfv applications. In *Proc. ACM SOSP '15* (2015), ACM.
[157] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., and Shenker, S. NetBricks: taking the V out of NFV. In *Proc. USENIX OSDI '16* (2016), USENIX.
[158] Park, V. D., and Corson, M. S. A highly adaptive distributed routing algorithm for mobile wireless networks. In *IEEE INFOCOM* (1997), vol. 3, pp. 1405–1413.
[159] Pfaff, B. Converging approaches in software switches. https://benpfaff.org/~blp/keynote.pdf, 2016.
[160] Pfaff, B., Pettit, J., Koponen, T., et al. The design and implementation of Open vSwitch. In *Proc. USENIX NSDI '15*.
[161] Phothilimthana, P. M., Liu, M., Kaufmann, A., Peter, S., Bodik, R., and Anderson, T. Floem: A programming system for NIC-accelerated network applications. In *Proc. USENIX OSDI '18* (2018), USENIX.
[162] Pongrácz, G., Molnár, L., Kis, Z. L., and Turányi, Z. Cheap silicon: A myth or reality? picking the right data plane hardware for software defined networking. In *Proc. ACM SIGCOMM HotSDN '13* (2013), ACM.
[163] Pontarelli, S., Bifulco, R., Bonola, M., et al. FlowBlaze: Stateful packet processing in hardware. In *Proc. USENIX NSDI '19* (2019).
[164] Popescu, D. A., Antichi, G., and Moore, A. W. Enabling fast hierarchical heavy hitter detection using programmable data planes. In *Proc. SOSR '17* (2017), ACM.
[165] Qazi, Z. A., Tu, C.-C., Chiang, L., et al. Simple-fying middlebox policy enforcement using sdn. *ACM SIGCOMM CCR 43*, 4 (2013).
[166] Rajagopalan, S., Williams, D., Jamjoom, H., and Warfield, A. Split/merge: System support for elastic execution in virtual middleboxes. In *Proc NSDI '13* (2013), nsdi'13, USENIX.
[167] Rétvári, G., Molnár, L., Pongrácz, G., and Enyedi, G. Dynamic compilation and optimization of packet processing programs. In *Proc. ACM SIGCOMM NetPL '17* (2017), ACM.
[168] Rétvári, G., Tapolcai, J., Kőrösi, A., Majdán, A., and Heszberger, Z. Compressing IP forwarding tables: Towards entropy bounds and beyond. In *Proc. ACM SIGCOMM '13* (2013), ACM.
[169] Rizzo, L. Netmap: a novel framework for fast packet I/O. In *Proc. USENIX ATC '12* (2012), USENIX.
[170] Rizzo, L., and Lettieri, G. Vale, a switched ethernet for virtual machines. In *Proc. ACM CoNEXT '12* (2012), ACM.
[171] Sanvito, D., Siracusano, G., and Bifulco, R. Can the network be the ai accelerator? In *SIGCOMM Workshop on In-Network Computing (NetCompute)* (2018), pp. 20–25.
[172] Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M., and Kalnis, P. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), ACM, pp. 150–156.
[173] Schiff, L., Schmid, S., and Kuznetsov, P. In-band synchronization for distributed sdn control planes. *ACM SIGCOMM Computer Communication Review (CCR) 46*, 1 (2016), 37–43.
[174] Schwartz, B., Jackson, A. W., Strayer, W. T., Zhou, W., Rockwell, R. D., and Partridge, C. Smart packets: applying active networks to network management. *ACM Transactions on Computer Systems (TOCS) 18*, 1 (2000), 67–88.
[175] SecuritytWeek. CSA's cloud adoption, practices and priorities survey report. http://www.securityweek.com/data-security-concerns-still-challenge, 2015. Accessed: 09-01-2019.
[176] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and implementation of a consolidated middlebox architecture. In *Proc. USENIX NSDI '12*.
[177] Sezer, S., Scott-Hayward, S., Chouhan, P. K., et al. Are we ready for SDN? Implementation challenges for software-defined networks. *IEEE Comm. Magazine 51*, 7 (2013).
[178] Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N., and Rexford, J. PISCES: a programmable, protocol-independent software switch. In *Proc. SIGCOMM '16* (2016), ACM.
[179] Shahbaz, M., and Feamster, N. The case for an intermediate representation for programmable data planes. In *Proc. SOSR '15* (2015), ACM.
[180] Sharma, N. K., Kaufmann, A., Anderson, T., Krishnamurthy, A., Nelson, J., and Peter, S. Evaluating the power of flexible packet processing for network resource allocation. In *Proc. USENIX NSDI '17* (2017), USENIX.
[181] Sharma, N. K., Liu, M., Atreya, K., and Krishnamurthy, A. Approximating fair queueing on reconfigurable switches. In *Proc. USENIX NSDI '18* (2018), USENIX.
[182] Shinde, P., Kaufmann, A., Roscoe, T., and Kaestle, S. We need to talk about NICs. In *Proc. USENIX HotOS '13*.
[183] Shrivastav, V. Fast, scalable, and programmable packet scheduler in hardware. In *Proc. ACM SIGCOMM '19*.
[184] Siracusano, G., and Bifulco, R. In-network neural networks. *CoRR abs/1801.05731* (2018).
[185] Sivaraman, A., Cheung, A., Budiu, M., et al. Packet transactions: High-level programming for line-rate switches. In *Proc. ACM SIGCOMM '16* (2016), ACM.
[186] Sivaraman, A., Subramanian, S., Alizadeh, M., et al. Programmable packet scheduling at line rate. In *Proc. ACM SIGCOMM '16* (2016), ACM.

[187] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. Heavy-hitter detection entirely in the data plane. In *Proc. ACM SOSR '17* (2017), ACM.
[188] Sonchack, J., Michel, O., Aviv, A. J., Keller, E., and Smith, J. M. Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With *Flow. In *Proc. USENIX ATC '18* (2018), USENIX.
[189] Srinivasan, V., Suri, S., and Varghese, G. Packet classification using tuple space search. In *Proc. ACM SIGCOMM '99* (1999), ACM.
[190] Stevens, W. P., Myers, G. J., and Constantine, L. L. Structured design. *IBM Systems Journal 13*, 2 (1974), 115–139.
[191] Stratum project. Developing an open source reference implementation for white box switches supporting next-generation SDN interfaces, 2018. https://stratumproject.org.
[192] Sun, C., Bi, J., Zheng, Z., et al. NFP: enabling network function parallelism in NFV. In *Proc. ACM SIGCOMM '17*.
[193] Thimmaraju, K., Hermak, S., Retvari, G., and Schmid, S. MTS: Bringing Multi-Tenancy to Virtual Networking. In *Proc. USENIX ATC '19* (2019), USENIX.
[194] Thimmaraju, K., Shastry, B., Fiebig, T., Hetzelt, F., Seifert, J.-P., Feldmann, A., and Schmid, S. The vamp attack: Taking control of cloud systems via the unified packet parser. In *Proc. CCS Workshop* (2017).
[195] Thimmaraju, K., Shastry, B., Fiebig, T., Hetzelt, F., Seifert, J.-P., Feldmann, A., and Schmid, S. Taking control of sdn-based cloud systems via the data plane. In *Proc. ACM SOSR '18* (2018), ACM.
[196] Tootoonchian, A., Panda, A., Lan, C., Walls, M., Argyraki, K., Ratnasamy, S., and Shenker, S. ResQ: enabling SLOs in network function virtualization. In *USENIX NSDI* (2018), pp. 283–297.
[197] Uzmi, Z. A., Nebel, M., Tariq, A., Jawad, S., Chen, R., Shaikh, A., Wang, J., and Francis, P. Smalta: practical and near-optimal fib aggregation. In *Proc. CoNEXT '11* (2011), ACM.
[198] Verdú, J., Nemirovsky, M., García, J., and Valero, M. Workload characterization of stateful networking applications. In *6th International Symposium on High Performance Computing* (2008), ISHPC, Springer Berlin Heidelberg.
[199] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P. Maple: simplifying sdn programming using algorithmic policies. *ACM SIGCOMM CCR 43*, 4 (2013), 87–98.
[200] Wang, H., Soulé, R., Dang, H. T., Lee, K. S., Shrivastav, V., Foster, N., and Weatherspoon, H. P4fpga: A rapid prototyping framework for p4. In *Proc. ACM SOSR '17* (2017), ACM.
[201] Wischik, D., Handley, M., and Braun, M. B. The resource pooling principle. *ACM SIGCOMM CCR 38*, 5 (2008).
[202] Woo, S., Sherry, J., Han, S., et al. Elastic scaling of stateful network functions. In *Proc. USENIX NSDI '18*.
[203] Xilinx. Vivado high-level synthesis. https://www.xilinx.com/products/design-tools/vivado.html.
[204] Yang, T., Xie, G., Liu, A. X., et al. Constant ip lookup with FIB explosion. *IEEE/ACM TON 26*, 4 (2018).
[205] Yu, M., Jose, L., and Miao, R. Software Defined Traffic Measurement with OpenSketch. In *Proc. USENIX NSDI '13*.
[206] Yuan, Y., Lin, D., Alur, R., and Loo, B. T. Scenario-based programming for SDN policies. In *Proc. ACM CoNEXT '15*.
[207] Zheng, Z., Bi, J., Wang, H., Sun, C., Yu, H., Hu, H., Gao, K., and Wu, J. Grus: Enabling latency SLOs for GPU-accelerated NFV systems. In *IEEE ICNP* (2018), pp. 154–164.
[208] Zhou, D., Fan, B., Lim, H., Kaminsky, M., and Andersen, D. G. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proc. CoNEXT '13* (2013), ACM.
[209] Zilberman, N., Audzevich, Y., Covington, G. A., and Moore, A. W. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro 34*, 5 (2014), 32–41.
[210] Zilberman, N., Watts, P. M., Rotsos, C., and Moore, A. W. Reconfigurable network systems and software-defined networking. *Proceedings of the IEEE 103*, 7 (July 2015), 1102–1124.