A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems

Roberto Bifulco
NEC Laboratories Europe, Germany
Email: roberto.bifulco@neclab.eu

Gábor Rétvári
MTA-BME Information Systems Research Group
Email: retvari@tmit.bme.hu

***Abstract*—Programmable switches allow the packet processing behavior to be applied to transmitted packets, including the type, sequence, and semantics of processing operations, to be reconfigured on the fly in a systematic fashion. As such, programmable switches are the key to realize the next-generation of network services and applications, including software-defined networking, 5G, IoT, and massive-scale cloud computing. This paper presents a survey on the recent trends and issues in the design and implementation of programmable network devices, focusing on the prominent abstractions and architectures proposed, debated, realized, and deployed during the last 10 years. First we describe the anatomy of a programmable switch, then we highlight the most important pointers from the literature and cast different taxonomies for the field, and finally we sketch open issues and possible future research directions.**

***Index Terms*—software-defined networks, data plane architecture, router design, programmable switches, data flow graphs, match-action pipelines, stateless and stateful network functions**

I. Introduction

With the advent of the 5G mobile standard, large-scale cloud computing, ubiquitous IoT, and massive machine learning and big data applications, operators will need to adopt completely new ways to architect communication networks, making software-defined networking, edge computing, network-function virtualization, and service chaining the norm rather than the exception. The forthcoming applications require Network Interface Cards (NICs) and network devices, such as switches and routers, to support continuously evolving and heterogeneous sets of protocols and functions on top of the impressive selection of features already supported today, including basic L2/L3/L4 processing, tunneling and VPN protocols, load balancing, congestion control and Quality of Service, firewalls and intrusion detection systems, etc.

Supporting such an extensive feature set at the required flexibility, dynamicity, performance, and efficiency requires careful and expensive engineering effort on the part of device vendors, usually involving the tedious and costly design, manufacturing, testing, and deployment of dedicated hardware components [1, 2]. This state of facts raises two problems. On the one hand, adding new features necessitates a long development process and rapid device upgrade cycles. This pushes vendors to support a given feature only when it becomes widely requested, impeding innovation. On the other hand, implementing every single network protocol in the packet processing logic leads to inefficiencies, due to wasting valuable memory space, CPU cycles, and silicon "real estate" for features that may never be used in operation.

To address these issues, a new generation of networking devices has been recently introduced, permitting the packet processing functionality implemented by a device to be comprehensively reconfigured on the fly. 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 [3, 4, 5, 6, 7]. Leveraging advances in IO frameworks [8, 9], these programmable software switches can achieve a forwarding throughput in the order of tens of Gbps on a single commodity server blade. More challenging workloads, in the range of hundreds of Gbps, are instead the realm of programmable hardware components and the devices built on top, like programmable NICs (SmartNICs) [10, 11, 12, 13] and programmable switches [14, 15, 16]. Similarly to software switches, programmable hardware-accelerated network appliances also offer various primitives that can be systematically assembled into complex network functions, using a domain specific language [17] or some dialect of a general purpose language [18, 19].

How to fix 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? These are currently among the most actively debated questions in the networking community. Following the footsteps of [20], in this paper we provide a survey on the current trends and issues in programmable software and hardware network devices, including the available abstractions, employed design solutions, and open problems. 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 software-defined networking, and the intellectual road that led there, see [21, 22, 23, 24].

The rest of the paper is organized as follows. In Section II we describe the most important components and functionality of a programmable switch and then in Section III we present the designs from the related literature organized according to

different taxonomies. In Section IV we highlight some of the most compelling issues and open problems in the field and finally in Section V we conclude the paper.

II. The Anatomy of a Programmable Switch

Conventional network gear, regardless of the implementation (e.g., pure software or specialized hardware) and function (e.g., a switch, an edge router, or a firewall), usually includes the control plane and the data plane integrated into a single device form factor. In this context, the 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 operations, including checking the device's health and performing maintenance operations. The data plane, in contrast, is responsible solely for executing the packet processing policy set by the control plane.

A fundamental enabler for network device programmability was the logical and physicial separation of the control plane from the data plane, with a standard and open API to allow for interactions between the two [24, 40, 36, 41]. In the case of OpenFlow, for instance, the data plane is realized by "dumb" OpenFlow switches administered remotely by a separate controller [36]. In this paper our focus is on these "dumb" switches and, in particular, the way they provide programmability, achieve high performance, and expose embedded state information to the control plane.

To enforce control plane decisions, the data plane performs a handful of operations on received packets. First, it parses (a subset of) the packet bytes to understand the control information they carry. Then, the extracted information is used to determine the sequence of processing operations that need to be applied to the packet, like performing some computation (e.g., calculation of a checksum), writing some parts of the packet (e.g., writing the new checksum into the appropriate header field), or storing monitoring and state information (e.g., updating a counter if the checksum was wrong). Finally, the modified packet is forwarded based on the results of the performed operations, which usually include the specification of the output port(s).

In this setting, *programmability refers to the capability of a switch to expose the packet processing logic to the control plane to be systematically, rapidly, and comprehensibly reconfigured*. The emphasis here is on the comprehensiveness; while conventional "fixed-function" network devices allow modification of the forwarding policy at least partially (e.g., adding static IP routes or changing ACLs), the ability to influence the parsing of packet headers, to provision forwarding tables that match on arbitrary header fields, or to introduce new processing actions, all in all, to facilitate the deployment of completely new network protocols in operation, is unique to programmable switches.

III. Taxonomies for Programmable Switches

In this section, we present broad classifications for the different architectural aspects of programmable data plane technologies. First, we start with a layer-oriented and an implementation-driven approach, and then we describe a programming-language-centered classification. Finally, we describe the stateless and the stateful data plane abstractions. The taxonomies are summarized in Table I. 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 [42].

A. System Level View

Embracing a systems-engineering approach, first we present a layered view of programmable switch architectures.

At the upper layer, the *control plane* is in charge of orchestrating the operations of the network as a whole. Traditionally, this entailed the specification of exhaustive per-switch control programs to set device-level behavior [43, 36]. With the fulfillment of the software-defined networking paradigm, however, the control plane has gradually moved towards higher-level abstractions that synthetically capture network-wide policies (the so called "intent layer"), which are then compiled into conventional device-level policies ("intermediate representations") in a separate step [19]. Examples of such high-level languages include Pyretic/Frenetic [25], Maple [26], Kinetic [27], NetEgg [28]; see [44] for a recent survey.

At the lower layer, the network's *data plane* is collectively represented by the packet processing logic of the programmable switches administered jointly by the control plane. The data plane exposes certain *abstractions* of its configurable functionality to the control plane which in turn uses data plane *programming languages* to configure the static packet processing semantics. At runtime this pipeline is filled with dynamic policies, e.g., entries are added to flow tables, actions are associated with flows, etc. [41, 45]. In addition, state and monitoring information is constantly collected in the data plane and presented to the control plane for verifying correct behavior, monitoring performance, and handling failures.

In this layered view, data plane compilation is a downwards mapping from a higher-layer description of the intended network behavior to lower-layer abstractions [36, 17, 25, 26, 27, 28, 44, 46], verification/monitoring is an upwards mapping whereby the lower-layer exposes some aggregate view of its state to the upper layers for checking and validation [27, 47, 48], and the coupling between the two is created by the abstractions adopted by the data plane. Interestingly, this gives rise to a circular dependency in the co-evolution of the control plane and the data plane, in that the capabilities of the data plane determine the elemental constructs high-level network programming languages are built upon, which in turn guide the data plane designers towards new processing abstractions (see e.g., the recent advances from OpenFlow[36] to P4 [17] and PoF [49] brought about by RMT [50, 51]).

B. Software vs. Hardware-based Switches

Processing each and every single packet received by the device, the data plane is the most performance-critical part of a network and, as such, usually specialized hardware components and sophisticated software acceleration methods are employed to implement it. On the hardware front the data plane

LAYER-BASED	CONTROL/INTENT LAYER Pyretic [25], Maple [26], Kinetic [27], NetEgg [28]	DATA PLANE LAYER Barefoot Tofino [16], NetFPGA [12], Cavium XPliant [14], Open vSwitch [3], BESS [29], Vpp [30], ESwitch [7], PISCES [31], NetBricks [32], AccelNet [33], Andromeda [34]
IMPLEMENTATION-BASED	SOFTWARE SWITCHES OVS [3], BESS [29], VPP [30], ESwitch [7], PISCES [31], NetBricks [32], Andromeda [34]	HARDWARE SWITCHES Barefoot Tofino [16], NetFPGA [12], Cavium XPliant [14], Intel Flexpipe [15], Netronome AgilioTM CX [10], Intel XScale [11]
ABSTRACTION-BASED	DATA FLOW GRAPH Click [35], VPP [30], BESS [29], NetBricks [32]	MATCH-ACTION PIPELINE OpenFlow [36], P4 [17], FAST [37], OpenState [38], Domino [39]
STATE-BASED	STATEFUL FAST [37], OpenState [38], NetBricks [32], Domino [39]	STATELESS OpenFlow [36], P4[1] [17]

TABLE I
TAXONOMIES FOR PROGRAMMABLE DATA PLANE DEVICES AND HIGHLIGHTS FROM THE RELATED LITERATURE.

functionality may be implemented in an ASIC (Application Specicic IC) [15, 16], an FPGA (Field-programmable Gate Array) [12, 33], or a network processor [14, 10, 11], using a dedicated packet classification engine realized in Ternary Content Addressable Memory chips (TCAM, [52]), whereas a software switch executes the entire processing logic on a commodity CPU [3, 29, 6, 30, 7, 31, 32, 34] on top of a fast packet-classification algorithm/data structure [53, 54, 55].

Note, however, that the distinction between hardware and software switches may not be so sharp in reality. For instance, a hardware-based appliance may still invoke a general purpose CPU (the "slow path") to run certain functions that do not require high-performance or that are not supported natively by the underlying hardware. Similarly, a modern software switch also relies on domain-specific hardware assistance for efficiency reasons, like Data Direct I/O (DDIO), segmentation offload (TSO/GSO), Receive Side Scaling and Receive Packet Steering (RSS/RPS), etc.

C. Abstractions and Architectures

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 access these primitives. Given this inherent architectural coupling, we next discuss the following three fundamental categories for programmable switches: the data flow graph abstraction and related switch architectures, the match-action pipeline abstraction, and hybrid switch architectures that implement different combinations of the previous abstractions.

1) The data flow graph abstraction: Early efforts borrowed heavily from generic systems design [56, 57] and machine learning [58], adopting the data-flow graph abstraction to architect programmable switches [35]. 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 then can be freely re-used 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 [35]. 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, ACLs, etc. 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 switching (in the sense of selecting a direction out of several alternatives) or load balancing and queuing.

ClickOS [4], Vector Packet Processing (VPP) from the FD.io project [30], the Berkeley Extensible Software Switch (BESS, [29]), and NetBricks [32] 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, using the built-in Single-Instruction-Multiple-Data (SIMD) instruction sets of modern CPUs, results more efficient software implementations [59, 9, 60]. 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 [32].

The presence of user-defined functionality abstracted as data flow graph nodes gives a great flexibility and extendibility [4, 61]. At the same time, this great flexibility tends to make the resultant designs piecemeal, and heterogeneity complicates high-level network-wide abstractions and encumbers performance optimization [62].

2) The match-action pipeline abstraction: The match-action abstraction describes data plane programs using a sequence of lookup tables organized into a hierarchical structure [36, 17, 3, 31, 7]. 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 further flow tables. The programmer in turn 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 [45]. This has the benefit of exposing reconfigurable data plane functionality to operators using the familiar notions of *flows* identified by matching *rules* defined on certain header fields (an

abstraction extensively used in firewalls and ACLs), structured as a hierarchy of lookup tables (used by conventional fixed-function router ASICs to synthesize L2/L3/L4 pipelines).

The match-action abstraction was popularized for programming switches by the OpenFlow protocol [36], which in turn borrowed greatly from Ethane [43]. OpenFlow at 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 basically arbitrary (but fixed) header fields. Currently Open vSwitch remains the most popular OpenFlow software switch [3], using a universal flow-caching based datapath for implementing the match-action pipeline. This design was improved upon by ESwitch, introducing data plane specialization and on-the-fly template-based datapath compilation to achieve line-rate OpenFlow software switching.

OpenFlow, however, is limited to a strict set of header fields; Protocol-Oblivious Forwarding (POF, [49]) and P4 (Programming Protocol-independent Packet Processors, [17]) generalize the match-action framework to parsing/deparsing arbitrary header fields, flexible lookup tables with rich semantics, configurable control flow, and platform-specific extensions. Again, this development was made possible by advances in switching ASIC design, such as the Reconfigurable Match Tables abstraction that overcomes two limitations in OpenFlow ASICs by letting match-action tables to be defined on arbitrary header fields and extending the, previously rather limited, repertoire of packet processing actions (RMT, [50]). Lately, P4 and the accompanying hardware and software switch projects [15, 16, 31] have met with increasing enthusiasm from the side of device vendors, operators, and service providers [63, 64].

3) Hybrid switch abstractions: 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 [36]. Indeed, 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. Not surprisingly, hybrid software-hardware switch architectures that mix the above abstractions with fresh ideas from distributed systems and multi-processor design have emerged in great numbers recently. Being a massively parallel workload, packet processing lends itself readily to be implemented in multi-threaded hardware like Graphics Processing Units (GPUs, [59]), with the pros and cons being actively debated in the systems community [65, 66]. Alternatively, dRMT extends the strictly sequential match-action processing abstraction of RMT towards a more flexible architecture [51], Intel FlexPipe [15] introduces a generic pipeline architecure, while FlexNIC 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 [67].

D. Stateful vs Stateless

Crucially, the content of network devices' embedded memory may be accessed and modified in operation. For example, the result of a forwarding table lookup may need to be temporarily stored to assign the packet to an output queue at a later stage. Alternatively, packet counter may be incremented under certain conditions. Such state information can be classified in two coarsely defined categories: *packet state* (or "metadata") and *global state* [19]. Packet state is associated with a single packet, each packet has its own packet state, and such state exists only during processing the associated packet. Conversely, global state is associated with the device, it is global, and it persists across packets. While packet state is accessed and modified only from within the data plane, global state may be accessed by the control plane as well. For example, packet's ingress port metadata is set and read exclusively by the data plane in order to, e.g., enforce access control rules; flow table entries are set by the control plane and read by, but usually not modified from, the data plane (global state); while a Network Address Translation mapping (also a global state) may be modified by both the data plane (on creation) and the control plane (e.g., for replication, migration, or scale-out).

Based on these considerations, a programmable data plane technology that is limited to only read global state but never to actually write it is generally referred to as *stateless* [36], while a data plane that also admits writing global state from within is called *stateful* [68]. In general, stateless data planes are much simpler but offer limited functionality (e.g., cannot implement a NAT without invoking a middlebox), while stateful data planes are much more complex, because the embedded global state information must be carefully synchronized between switches that may simultaneously access them [69].

IV. Open Issues

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. In addition, an ideal abstraction should lend itself easily to be exposed to the control plane as a secure and efficient data plane API [41, 36], it should adequately handle global state embedded in the data plane and provide a well-defined consistency model [69], it should admit analytic performance models [70, 7] and automatic program transformations for performance optimization [7], it should separate static semantics from dynamic behavior [45], 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.

1) Abstractions for comprehensive reconfigurability: With the recent trends toward moving from the rigid data plane programming model of OpenFlow to the much more flexible P4 world, it has become essential to abstract and expose

every aspect of processing functionality a switch may perform. 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 [71] to universal scheduling and queuing schemes [72, 68].

But the quest towards more comprehensive reconfigurability at the data plane should not stop at this point; indeed, new abstractions for supporting multipath routing [73], dynamic load-balancing [74], and data-plane level isolation and encryption [32, 75], would open the door to interesting new *in-network computation* schemes to be deployed on top [76], like network-accelerated key-value stores [77] and distributed consensus protocols [78, 79], network-wide machine learning with data-plane neural networks [80], programmable monitoring and measurement [81], etc.

Open Problem 1. *Find novel abstractions to render further data plane functionality configurable and expose these abstractions to the control plane using a simple API.*

2) Abstractions for exposing global state: The need to scale systems to handle massive workloads increasingly pushes designers to explore more complex solutions that handle some state already in the switches' data plane [74, 82, 83]. While abstractions for stateless packet processing can be considered rather solid at this point in time, stateful abstractions are still in their infancy and no clear winner has yet emerged.

The complexity of a stateful abstraction lays in the need to address state management problems (e.g., consistency) in a programmer-friendly way, while still guaranteeing the ability to support high-performance implementations. Here, it should be noted that reading and writing to a memory is one of the main sources of performance issues in modern computing systems [50, 84]. Current proposals follow three different approaches: consistency-, executor- and application-based memory abstractions.

Consistency-based memory abstractions organize memory according to the required consistency properties. For instance, in NetBricks [32] a programmer can read and write state to a memory that guarantees strong consistency, eventual consistency, or no consistency at all. Some abstractions, may provide a subset of the consistency models, such as Domino [39] only provides strong consistency (at the cost of limiting the range of supported algorithms at line rate). In executor-based memory abstractions the programmer is exposed to a simplified view of the underlying executor architecture. packetC [18] follows this approach by dividing the memory in processor-local and global areas. Finally, application-based memory abstractions expose access to state following common programming abstractions found in network applications. This is the case of OpenState [38] and FAST [37], which in addition to global state define also flow state. An instance of flow state is associated with a single network flow and can be accessed only during the processing of the such flow.

The three approaches implement different trade-offs. For example, the executor-based abstraction gives the most flexibility to the programmer, at the cost of increasing programming complexity and reducing portability among executor architectures, since a programmer needs to have a clear understanding on how to partition its application's state among different memory areas. On the other side of the spectrum, application-based abstractions provide a state access model that is in line with most of the network applications, therefore simplifying the development for the programmer, but leaving little flexibility to experiment unconventional solutions.

A recent research direction tries to asses the effective need to keep state in the data plane, by proposing an architecture where a separate, dedicated data-store is in charge of maintaining data-plane state while the data plane itself is stateless [85, 69]. Accordingly, when a switch needs some state information it dynamically fetches it from the state-store. This architecture makes state explicit and factors it out from switches, opening the door to elastic scale-out, replication, migration, and restoration [69]. Nonetheless, it should be noted that some sort of soft state, i.e., a local cache, is usually still required for performance reasons [86]. The handling of this cache could be defined through one of the aforementioned abstractions.

Open Problem 2. *Find an expressive yet simple model to handle state operations in the data plane, simplifying application programming and orchestration while ensuring the ability of supporting high-performance implementations.*

3) Abstractions for intent-based networking: Intent-based networking marks the recent trend towards designing and operating networks in terms of higher-level business policies, and letting the network to deal with low-level concerns in an automated, agile, secure, and verifiable way [87]. Intent-based networking, however, raises a number of crucial problems in programmable switch design.

Although 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 [25, 26, 27, 28, 44, 46], 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 [7] with the minimal data plane footprint [88, 89], built on a firm theoretical foundation for optimizing data plane programs and reasoning about performance [70, 7].

Open Problem 3. *Design an optimizing intent-based data plane compiler to map high-level business policies (and the corresponding global state) to the underlying physical infrastructure, built on a sound theory for data plane program transformation and performance modeling.*

4) Abstractions for verifiability: Data plane compilation, that is, downward mapping from the intent layer to the data

plane is just one side of the coin. 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 [27], which may require the definition of new abstractions.

Open Problem 4. *Find data plane abstractions and build monitoring/verification frameworks on top, which allow the intent layer to discover operational context, including global state and load/resource availability, to verify correct operations and to uncover network performance anomalies potentially originating from malicious activity.*

5) Abstractions for security and scalability: The size and of the workloads that can be supported economically, efficiently, and securely with programmable switches is fundamentally constrained on the amount of resources available in the data plane. Hardware switches are evidently limited by the capacity of the ASICs and, in particular, the size of the built-in TCAMs [90], and software switches are heavily burdened by the inherent computational complexity of performing packet classification on general purpose CPUs [53, 55, 54]. Indeed, recent reports have found that the scalability and security [91] of the programmable data plane remain to be of concern today.

Open Problem 5. *Improve the general performance, scalability, and security of programmable switches.*

V. Conclusion

This paper provided a brief survey of the recent advancements in programmable data plane design and implementation, pointing the reader to a few relevant open problems. The field is currently under active development, with new findings and contributions appearing at a fast pace. To keep track of such development, we are building and updating an annotated online reading list [42], and encourage researchers to contribute.

Acknowledgment

This project has received funding from the European Unions Horizon 2020 research and innovation programme under the grant agreement No 671648 and 761493. The content of this paper does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s). G. R. is with the Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics.

References

[1] S. Sezer et al. “Are we ready for SDN? Implementation challenges for software-defined networks”. In: *IEEE Communications Magazine* 51.7, 2013.

[2] N. McKeown. “Programmable Forwarding Planes are Here to Stay”. In: NetPL 2017.

[3] B. Pfaff et al. “The Design and Implementation of Open vSwitch”. In: USENIX NSDI ’15.

[4] J. Martins et al. “ClickOS and the Art of Network Function Virtualization”. In: USENIX NSDI’14.

[5] L. Rizzo et al. “VALE, a Switched Ethernet for Virtual Machines”. In: ACM CoNEXT ’12.

[6] M. Honda et al. “mSwitch: A Highly-scalable, Modular Software Switch”. In: ACM SOSR ’15.

[7] L. Molnár et al. “Dataplane Specialization for High-performance OpenFlow Software Switching”. In: ACM SIGCOMM ’16.

[8] L. Rizzo. “Netmap: a novel framework for fast packet I/O”. In: USENIX ATC’12.

[9] Intel. *Intel DPDK: Data Plane Development Kit*. project website. 2016.

[10] Netronome. *AgilioTM CX 2x40GbE Intelligent Server Adapter*. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf.

[11] 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.

[12] N. Zilberman et al. “NetFPGA SUME: Toward 100 Gbps as Research Commodity”. In: *IEEE Micro ’14* 34.5, 2014.

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

[14] *XPliant Ethernet Switch Product Family*. http://www.cavium.com/XPliant-Ethernet-Switch-ProductFamily.html.

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

[16] Barefoot Networks. *Barefoot Tofino: World’s fastest P4-programmable Ethernet switch ASICs*. https://barefootnetworks.com/products/brief-tofino/.

[17] P. Bosshart et al. “P4: Programming protocol-independent packet processors”. In: *ACM SIGCOMM CCR* 44.3, 2014.

[18] R. Duncan et al. “packetC: Language for High Performance Packet Processing”. In: IEEE HPCC 2009.

[19] M. Shahbaz et al. “The Case for an Intermediate Representation for Programmable Data Planes”. In: ACM SOSR ’15.

[20] S. Keshav et al. “Issues and trends in router design”. In: *IEEE Communications Magazine* 36.5, May 1998.

[21] N. Zilberman et al. “Reconfigurable Network Systems and Software-Defined Networking”. In: *Proceedings of the IEEE* 103.7, 2015.

[22] D. Kreutz et al. “Software-defined networking: A comprehensive survey”. In: *Proceedings of the IEEE* 103.1, 2015.

[23] B. A. A. Nunes et al. “A survey of software-defined networking: Past, present, and future of programmable networks”. In: *IEEE Communications Surveys & Tutorials* 16.3, 2014.

[24] N. Feamster et al. “The Road to SDN”. In: *Queue* 11.12, Dec. 2013.

[25] C. Monsanto et al. “Composing Software Defined Networks”. In: USENIX NSDI 2013.

[26] A. Voellmy et al. “Maple: simplifying SDN programming using algorithmic policies”. In: *ACM SIGCOMM CCR* 43.4, 2013.

[27] H. Kim et al. “Kinetic: Verifiable Dynamic Network Control”. In: NSDI’15.

[28] Y. Yuan et al. “Scenario-based Programming for SDN Policies”. In: CoNEXT ’15.

[29] S. Han et al. *SoftNIC: A Software NIC to Augment Hardware*. unpublished manuscript. 2015.

[30] FD.io. *The Fast Data Project*. project website. 2016.

[31] M. Shahbaz et al. “PISCES: A Programmable, Protocol-Independent Software Switch”. In: ACM SIGCOMM ’16.

[32] A. Panda et al. “NetBricks: Taking the V out of NFV”. In: USENIX OSDI’16.

[33] D. Firestone et al. “Azure Accelerated Networking: SmartNICs in the Public Cloud”. In: USENIX NSDI 2018.

[34] M. Dalton et al. “Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization”. In: USENIX NSDI 2018.

[35] R. Morris et al. “The Click modular router”. In: ACM Trans. on Computer Systems 2000.

[36] N. McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *ACM SIGCOMM CCR* 38.2, Mar. 2008.

[37] M. Moshref et al. “Flow-level State Transition As a New Switch Primitive for SDN”. In: ACM HotSDN ’14.

[38] G. Bianchi et al. “OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch”. In: *ACM SIGCOMM CCR* 44.2, Apr. 2014.

[39] A. Sivaraman et al. “Packet Transactions: High-Level Programming for Line-Rate Switches”. In: ACM SIGCOMM ’16.

[40] L. Yang et al. *Forwarding and Control Element Separation (ForCES) Framework*. RFC 3746. IETF. Apr. 2004.

[41] P4.org. *P4 Runtime*. https://p4.org/p4-runtime.

[42] R. Bifulco et al. *The Programmable Data Plane: Reading List*. https://rg0now.github.io/prog_dataplane_reading_list/README.html.

[43] M. Casado et al. “Ethane: Taking Control of the Enterprise”. In: ACM SIGCOMM ’07.

[44] N. Foster et al. “Languages for software-defined networks”. In: *IEEE Communications Magazine* 51.2, 2013.

[45] G. Rétvári et al. “Dynamic Compilation and Optimization of Packet Processing Programs”. In: ACM SIGCOMM NetPL 2017.

[46] L. Jose et al. “Compiling Packet Programs to Reconfigurable Switches”. In: USENIX NSDI ’15.

[47] P. Kazemian et al. “Real Time Network Policy Checking Using Header Space Analysis”. In: USENIX NSDI 13.

[48] A. Khurshid et al. “Veriflow: Verifying network-wide invariants in real time”. In: ACM SIGCOMM HotSDN 2012.

[49] H. Song. “Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane”. In: ACM HotSDN ’13.

[50] P. Bosshart et al. “Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN”. In: ACM SIGCOMM ’13.

[51] S. Chole et al. “dRMT: Disaggregated Programmable Switching”. In: ACM SIGCOMM ’17.

[52] P. K. et al. “Content-addressable memory (CAM) circuits and architectures: A tutorial and survey”. In: *IEEE Journal of Solid-State Circuits* 41.3, 2006.

[53] A. Feldman et al. “Tradeoffs for packet classification”. In: IEEE INFOCOM 2000.

[54] K. Kogan et al. “SAX-PAC (Scalable And eXpressive PAcket Classification)”. In: ACM SIGCOMM ’14.

[55] V. Srinivasan et al. “Packet Classification Using Tuple Space Search”. In: ACM SIGCOMM ’99.

[56] W. P. Stevens et al. “Structured design”. In: *IBM Systems Journal* 13.2, 1974.

[57] I. D. Zone. *Data Flow Graph*. Web page, last accessed April 20 2018. 2018.

[58] M. Abadi et al. “TensorFlow: A system for large-scale machine learning”. In: USENIX OSDI 2016.

[59] S. Han et al. “PacketShader: A GPU-accelerated Software Router”. In: ACM SIGCOMM ’10.

[60] T. Barbette et al. “Fast Userspace Packet Processing”. In: ANCS ’15.

[61] R. Laufer et al. “CliMB: Enabling Network Function Composition with Click Middleboxes”. In: HotMIddlebox ’16.

[62] B. Li et al. “ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware”. In: ACM SIGCOMM ’16.

[63] N. Kumar. *Juniper Advancing Disaggregation Through P4 Runtime Integration*. https://forums.juniper.net/t5/The-New-Network/Juniper-Advancing-Disaggregation-Through-P4-Runtime-Integration/ba-p/319195. 2018.

[64] Stratum project. *Developing an open source reference implementation for white box switches supporting next-generation SDN interfaces*. https://stratumproject.org. 2018.

[65] A. Kalia et al. “Raising the Bar for Using GPUs in Software Packet Processing”. In: USENIX NSDI’15.

[66] Y. Go et al. “APUNet: Revitalizing GPU as Packet Processing Accelerator”. In: USENIX NSDI 2017.

[67] A. Kaufmann et al. “High Performance Packet Processing with FlexNIC”. In: *SIGPLAN Not.* 51.4, Mar. 2016.

[68] A. Sivaraman et al. “Programmable Packet Scheduling at Line Rate”. In: ACM SIGCOMM ’16.

[69] S. Woo et al. “Elastic Scaling of Stateful Network Functions”. In:

[70] M. Bansal et al. “Atomix: A Framework for Deploying Signal Processing Applications on Wireless Infrastructure”. In: USENIX NSDI 2015.

[71] G. Gibb et al. “Design principles for packet parsers”. In: ANCS ’13.

[72] R. Mittal et al. “Universal Packet Scheduling”. In: USENIX NSDI 2016.

[73] S. Ghorbani et al. “DRILL: Micro Load Balancing for Low-latency Data Center Networks”. In: SIGCOMM ’17.

[74] R. Miao et al. “SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs”. In: ACM SIGCOMM ’17.

[75] R. Poddar et al. “SafeBricks: Shielding Network Functions in the Cloud”. In: USENIX NSDI 2018.

[76] G. Bianchi et al. “Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing”. In: *CoRR* abs/1605.01977, 2016.

[77] X. Li et al. “Be Fast, Cheap and in Control with SwitchKV”. In: USENIX NSDI 2016.

[78] H. T. Dang et al. “Paxos Made Switch-y”. In: *ACM SIGCOMM Comput. Commun. Rev.* 46.2, May 2016.

[79] H. T. Dang et al. “NetPaxos: Consensus at Network Speed”. In: ACM SOSR ’15.

[80] G. Siracusano et al. “In-network Neural Networks”. In: *CoRR* abs/1801.05731, 2018. arXiv: 1801.05731.

[81] Y. Gong et al. “Towards Accurate Online Traffic Matrix Estimation in Software-defined Networks”. In: ACM SOSR ’15.

[82] X. Jin et al. “NetCache: Balancing Key-Value Stores with Fast In-Network Caching”. In: SOSP ’17.

[83] N. K. Sharma et al. “Evaluating the Power of Flexible Packet Processing for Network Resource Allocation”. In: USENIX NSDI 2017.

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

[85] M. Kablan et al. “Stateless Network Functions: Breaking the Tight Coupling of State and Processing”. In: USENIX NSDI 2017.

[86] T. Dietz et al. “Enhancing the BRAS through virtualization”. In: IEEE NetSoft 2015.

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

[88] G. Rétvári et al. “Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond”. In: ACM SIGCOMM ’13.

[89] A. X. Liu et al. “TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs”. In: *IEEE/ACM Trans. Netw.* 18.2, Apr. 2010.

[90] M. Kuzniar et al. *What you need to know about SDN control and data planes*. Tech. rep. EPFL-REPORT-199497. EPFL, 2014.

[91] T. Dargahi et al. “A Survey on the Security of Stateful SDN Data Planes”. In: *IEEE Communications Surveys Tutorials* 19.3, 2017.