P4DNS: In-Network DNS

Jackson Woodruff
University of Cambridge
Cambridge, UK
J.C.Woodruff@sms.ed.ac.uk

Murali Ramanujam
University of Cambridge
Cambridge, UK
murali.ramanujam@cl.cam.ac.uk

Noa Zilberman
University of Cambridge
Cambridge, UK
noa.zilberman@cl.cam.ac.uk

***Abstract*—In-network computing offers an appealing scalability trajectory for network services, as application performance scales with network devices. Despite its potential, in-network computing may not be suitable for all applications, due to paradigm assumptions and network-device limitations. As users' Internet demands keep growing, any limitations on the scalability of network services such as DNS limits the scalability of end-to-end experience. In this paper we present P4DNS, an in-network DNS solution, exploring the span and limitations of implementing a realistic network service within a network device using P4. P4DNS is a high performance DNS server, implemented in P4 over NetFPGA and providing $\times 52$ performance improvement compared with software-based solutions. P4DNS provides insight into the limitations of implementing in-network services using today's paradigms, and the trade-offs between data and control planes.**

I. Introduction

The demand for moving and processing data is growing. More than 10 zettabytes of data are processed every year — a number expected to double by 2021 [1]. The increasing demands burden both communication and computing infrastructure, as a significant number of network services are associated with today's communications.

Programmable network devices have opened a road to scalable network services [2]. Processing data within network devices offloads processing from servers, freeing clock cycles, and achieves extremely high performance at the scale of billions-of-operations per second [3]. Furthermore, moving applications to the network reduces latency, which benefits latency-sensitive applications [4]. While many works focus on caching [3], [5], machine learning [6], [7] and stream processing [8], network services are not ignored [2], [9].

Despite the impressive performance figures, cloud providers have been hesitant to deploy such solutions [6]. Further work discussed the limitations and challenges in deploying in-network computing [10], [11]. Still, there has been little detailed discussion on the architectural limitations of the programmable data planes paradigm on the implementation of real-world applications. For example, Tokusashi *et al.* [5] studied the implications of design decisions on the performance and power of a caching application, but did not discuss functional limitations.

We focus on one specific limitation of programmable data planes — the separation of control and data planes — and the entailed separation of functionality. While the roots of this separation are in software-defined networks (SDN) [12], most commercial products [13] maintain this separation for strong isolation, better manageability, and development simplification [14]. Stateful data plane tasks require specialized hardware [15] or user-defined modules (e.g., externs [16]).

We use one common network service, a domain name server (DNS), as the leading use-case of our work. We present P4DNS: a DNS implemented on a programmable platform, NetFPGA-SUME [17], written in P4 [16], using the standard P4-NetFPGA workflow [18]. While previous work [2] has focused on the ability to reply to a DNS query, we explore the limitations of deploying the complete service within a network device. While our design achieves $\times 52$ performance improvement compared with a software-based solution, we show that the separation of planes limits the potential scalability of network services deployed within network devices. Our code can be found at https://github.com/cucl-srg/P4DNS.

In summary, this work makes the following contributions:

- We introduce the hardware/software architecture of P4DNS — a transparent, in-network DNS.
- We describe P4DNS's implementation of a P4-based DNS data plane architecture.
- We present P4DNS's control plane, designed to support high-rate updates.
- We discuss the limitations of data and control plane separation — in particular that mutable state is managed from the control plane, and how the P4 language and its supporting architectures affect P4DNS's design.

This paper is organized as follows: Section II describes DNS. Section III describes P4DNS's architecture. Sections IV and V describe the design decisions in P4DNS's data and control planes respectively. In Section VI we benchmark P4DNS. Finally, sections VII and VIII discuss P4DNS as a whole and how it fits into the related work. We conclude in Section IX.

II. Background

DNS is a critical service for the Internet to map user-friendly domain names to machine-friendly IP addresses. Each such mapping is called an "A record" and is stored in a name server. A domain name is a series of labels each of up to 255 octets. In addition, each record contains meta-data, such as the remaining time the record is valid.

DNS records are accessed using DNS requests and DNS responses. A DNS request specifies a number of *questions* and elicits a DNS response specifying a number of *responses*.

Request ID				
Question or Response	Opcode	Flags	Unused	Response Code
Query Count				
Answer Count				
Name Server Record Count				
Additional Records Count				
Question/Answer Headers (Optional)				
...				

Fig. 1. DNS packet structure. Each row is 16 bits long.

Fig. 1 illustrates the format of a DNS packet. Request and response packets use the same structure, and questions and responses are appended to the DNS header. DNS records offer compression for domain names. Each requested name only need appear once in full. Subsequent occurrences can be replaced by a pointer to the first occurrence.

DNS achieves scalability using a hierarchy. Top-level name servers store authoritative copies of DNS records. Lower-level name servers cache records. When a low-level DNS server receives a question it cannot answer, there are two ways to resolve the request. Requests may be *iteratively* resolved, in which case the DNS server responds with the address of another name server for the requester to query, or they may be *recursively* resolved, in which case the low-level server sends the question to another name server. The type of resolution is controlled with the "recursion desired" flag. Further details can be found in RFC 1034 [19] and RFC 1035 [20].

III. Architecture

P4DNS behaves similarly to a low-level name server while enabling seamless integration into data center networks. To achieve seamless integration, P4DNS needs no IP address like traditional DNS servers, but snoops and responds to DNS requests passing through a switch. For requests without a cached responses, P4DNS uses recursive resolution if requested, and otherwise forwards the response as a switch.

As an in-network computing DNS, P4DNS builds on the programmability of data planes. However, P4DNS is not limited to the data plane alone, and uses a converged data and control plane architecture to achieve higher performance. P4DNS achieves $\times 52$ NSD's [21] and $\times 10$ Emu's [2] throughput, with no packet loss. It maintains low latency, and the difference between median and tail latency is ± 30ns for cached entries.

P4DNS's architecture supports the following features:

1) Packet switching functionality for non-DNS records.
2) Response to A record queries of multiple lengths.
3) Update of the DNS cache when the switch passively observes a DNS response on the wire.
4) Maintenance of the TTL of each cached domain name.
5) Recursive resolution with a multi-threaded control plane.

P4DNS's architectural decisions are driven by the question: "How much can be done within a match-action pipeline?" Components that fit poorly into a match-action pipeline require the control plane, which quickly becomes a bottleneck.

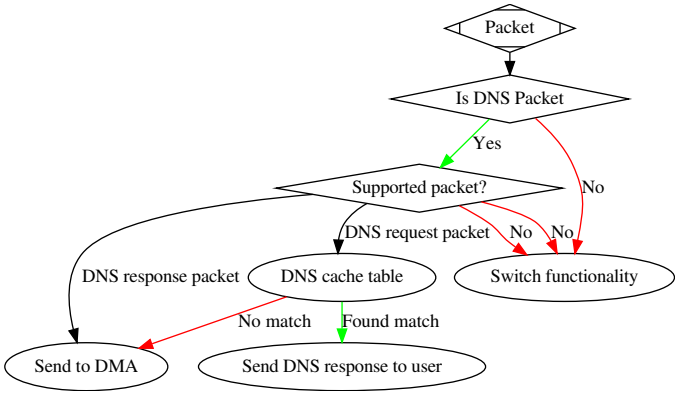

Fig. 2. P4DNS data plane state transitions.

P4DNS's architecture is based on the design of, and maintains the behavior of, a P4-based Ethernet switch. We envision our architecture used in data center ToR switches, accelerating DNS requests performed by many machines in the same rack.

The control plane implements important features of our switch. The control plane handles learning behavior, recording which ports MAC addresses correspond to. Beyond learning, direct memory access (DMA) is used to send two types of packets to the control plane: recursive DNS requests and uncached DNS responses. The control plane spawns a new thread to handle each packet it receives with DMA. The control plane also manages an internal cache table with each seen domain name and its TTL. It updates these TTLs every second and sends these updated values to the data plane. Finally, the control plane tracks the number of entries in the data plane cache to prevent it from overflowing. All this functionality exists in the control plane because it involves *managing state*, which is not possible in the data plane without complicated, proprietary and difficult to port P4 externs.

The data plane handles P4DNS's performance-critical components. It responds to DNS requests when possible and acts as a switch when not. This is done using the parser and a series of condition checks in the main action pipe. If the packet is a DNS request, an exact match table is used to lookup the name. Because of the vastly higher efficiency of the data plane, it is best suited to quickly reply to queries.

We implement P4DNS on a programmable platform (NetFPGA SUME [17]). Our design is portable to most other hardware targets, as it uses no target-proprietary modules.

IV. Data Plane

P4DNS's uses a P4 data plane, designed to run on the NetFPGA SUME platform [17]. We use the Xilinx SDNet P4C compiler and build P4DNS on top of a reference P4 learning switch, using SimpleSumeSwitch architecture[1].

We include as much behavior as possible within the data plane to reduce the load on the software control plane. This section describes the components of the data plane, how desired behavior is represented in P4 tables, and limitations we encountered. There are three stages for each packet:

[1]Documentation for SimpleSumeSwitch is available at: https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Ethernet-Learning-Switch.

1) Parser: extracts important fields from the packet.
2) Main action: handles forwarding and DNS behavior.
3) Deparser: reconstructs the packet for it to be emitted.

A. Parser

The parser inspects every incoming packet and extracts the headers to the DNS layer. Parsing stops before non-existent headers are parsed in non-DNS packets. The parser must understand how the length of DNS headers changes as the requested domain length varies. However, the P4C compiler does not support parsing of variable length fields. To support variable length domain names, we use a different state for each supported packet length. To allow lookup of all names within a single table, shorter names are padded with zeroes.

B. Main Action

After the parser, the main action checks whether a DNS response (or other action) is required by checking header fields. This is done after parsing to avoid complexity in the parser which uses excessive hardware resources.

A flow diagram of P4DNS's behavior is shown in fig. 2. DNS requests are passed to the DNS table. The domain name in the DNS packet is matched into the table. If a result is found, a response packet is created by swapping the source and destination addresses and appending a DNS response header. DNS responses are forwarded as normal — DMA is used in parallel to send a copy to the control plane which updates the match action tables. When a packet with the recursion desired bit set misses the DNS table, it is *not forwarded*, but sent to the control plane using DMA to be resolved.

C. Deparser

The deparser is a mirror image of the parser. It emits all parsed packet headers. If response fields are added, the deparser also emits those. `if` conditions are not used in the deparser, headers emission is implicitly controlled using a validity bit that marks whether a field should be emitted.

D. Limitation and Challenges

We found that adding many states to the parser increased compile time and hardware resource usage beyond reasonable limits. We removed much boolean logic from the parser and into the main action to address this. Largely, this is a limitation of the parser model. Our parser does not require the full power of a non-recursive state machine provided by P4C. In fact, we could just extract the first 65 B except P4's behaviour when extracting off the end of a packet is undefined, and in practice means packets switching fails[2]. This problem could be addressed with appropriate compiler optimization: given the resource usage we suggest P4 compiler writers include such optimizations for targets where the parser overhead is high. However, relying on compiler optimizations is stopgap rather than a solution because small program changes can cause optimizations to silently fail. Alternatively, if lookahead parsing without failing on short packets was possible, the parsing of Ethernet, IPv4, UDP and DNS headers could be compressed into a single state.

[2]The failure mode for parsing off the end of a packet is to make all fields invalid. This safety and security feature leads to a parser that must have many states to avoid disabling the Ethernet switching functionality on packets that are short.

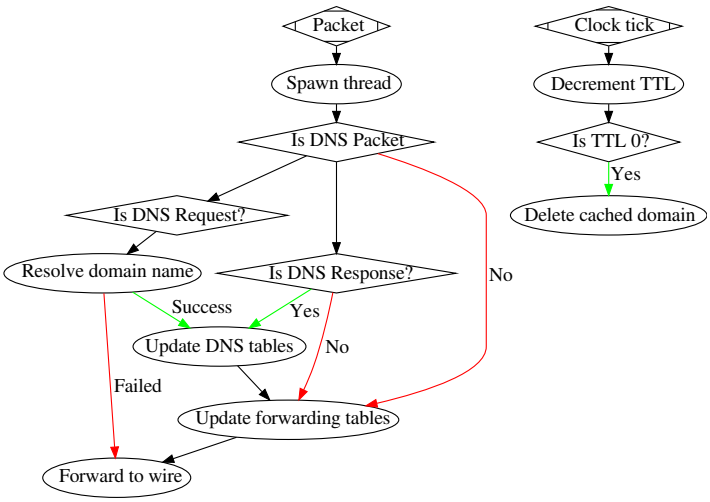

Fig. 3. The logic flow of the control plane of P4DNS.

Beyond the parser state problems, we discovered that some checks required by DNS do not fit the P4 model particularly well. For example, DNS names are designed for C-style for-loops, similar in concept to null terminated string. One parsing approach is using externs, which comes at the cost of portability. In our case, the packet length and several DNS header fields were sufficient to determine the requested domain name's length. However, other applications may not have this luxury.

V. Control Plane

The control plane is a highly parallel implementation in Python that spawns a new thread to handle each packet received via DMA. A flowchart of control plane behavior is shown in fig. 3. Each incoming message is either a digest containing a MAC address and source port for the learning switch, or a whole DNS packet that may either request for recursive resolution or a response to update the cache. For digests, the control plane updates the switch forwarding tables. If a whole packet is received and is a DNS response, the response is extracted and the local caches are updated. If a DNS request is received with the recursion desired bit set, the control plane recursively resolves the request. If a domain name is unresolvable, the control plane sends an appropriate DNS error response.

A. Cache Updates

A significant part of the control plane is dedicated to managing cache updates — identifying which DNS information to cache, and inserting it into the hardware cache tables. There are three opportunities to update the onboard caches:

- When a DNS response passes through the switch.
- When a recursive query is answered.
- When a TTL for a cached domain changes.

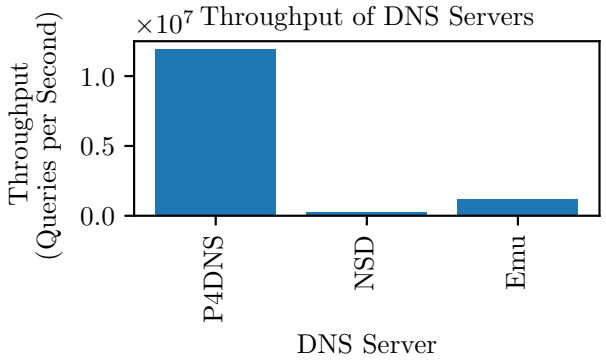

Fig. 4. Throughput comparison of DNS servers.

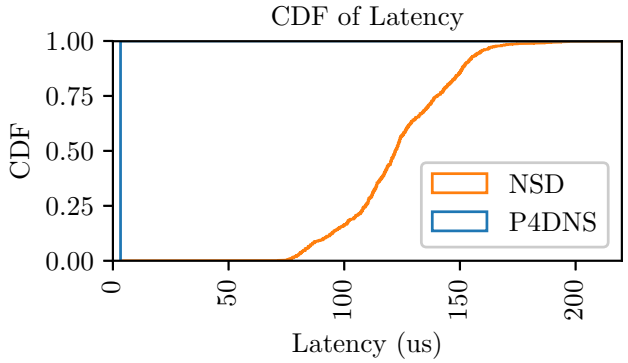

Fig. 5. P4DNS vs. NSD response latency CDF.

1) Table Models: These updates are managed using a model in the control plane of the P4 tables. The hardware cache in the prototype has 64 entries and is managed using a FIFO eviction protocol. When P4DNS's control plane caches a new response in a full hardware cache, an old response is evicted.

To keep the TTL of every DNS response accurate, a timer updates the TTL of all DNS records stored in both the host and onboard tables once a second. When a DNS record reaches a TTL of 0, it is removed from both caches.

B. Design Challenges

The concurrent implementation introduces bottlenecks where many threads compete for shared resources (P4 tables and table models). P4 table writes are not atomic in P4→NetFPGA, so are not thread safe. P4DNS uses a wrapper class with locks around SDNet calls to ensure thread safety. Using locks forces careful consideration to avoid deadlock. We follow the traditional approach of ensuring locks are always taken in the same order. However, crashes while holding locks still cause deadlock. For example, if we receive A records with a TTL of 0. Careful management of these records is required to ensure that they are not marked as expired and removed before a response packet is sent to the requesting host. To address this issue more generally, locks are acquired within `try`, `finally` blocks — locks are released if there is a crash, although data may remain in an inconsistent state.

The SDNet APIs presented challenges beyond concurrency. Attempting to insert entries into full hardware tables results in silent failure. Our table models (section V-A1) address this.

Python library Scapy [22] is used to process packet data. We use a custom Scapy packet structure to process Ethernet digest information. When processing DNS packets, we found that packet length in the control plane varies from the actual length on the wire despite no visible content difference. This issue is related to DNS compression — a domain name appearing more than once in a packet can be replaced by a pointer to another occurrence. Scapy "helpfully" decompresses packets, resulting in larger sizes in Python than Wireshark. We patched Scapy in our control plane to prevent DNS decompression.

VI. Evaluation

In this section, we evaluate P4DNS's performance. We compare P4DNS's latency and throughput on cached responses to NSD [21] and Emu [2], and discuss the limitations of P4DNS's functionality compared with NSD and Emu.

A. Experimental Setup

Our experimental setup uses an ExaNIC HPT capture card [23] and an optical tap to measure latency and throughput of DNS servers. DNS requests entering the DNS server under test are mirrored to a capture card using an optical tap. The DNS response is also sent to the capture card. In this setup, the capture card receives one copy of each packet: one before the request enters the server and one as the reply is sent. The difference between arrival times indicates device latency.

OSNT [24], an open source network tester, is used to generate DNS packets. A single 10GE port is used to generate traffic s both Tx and Rx traffic can be captured on a (dual port) ExaNIC HPT. Where used, NSD runs on an Intel Xeon E5-2637 v4 machine running at 3.50 GHz with 64 GB of RAM and using a Solarflare SFC9220 NIC.

1) Throughput: To measure throughput, we generate 10 million 64 B DNS requests with OSNT, and calculate the throughput based on the time required to process 10 million 80 B replies. We find that P4DNS can respond to DNS requests at line rate (11 million responses per second at 10GE). Fig. 4 shows this is 52 times the performance of NSD (226,000 responses per second, on the same machines, as reported in [2]), and 10 times the performance of Emu, (1.1 million responses per second as reported in [2]).

2) Latency: To measure latency, we send individual DNS requests using `tcpdump` with a large inter-packet gap in to test DNS servers under no load. We used the same setup for both P4DNS and NSD (for Emu we used the results from [2]). All latency experiments were run with 1000 packets.

Latency tests show little difference between latency distributions of different sized queries hitting our cache. For P4DNS, The median response latency of cached responses is 3.33 µs (99th percentile 3.35 µs) for both 64 B and 65 B DNS request

packets. In NSD, the median latency is 122.25 µs and (99th percentile 181.73 µs).

To understand the effect of our underlying switch, we performed the same measurement on the learning switch without P4DNS. In this experiment, the median latencies are lower, at 1.675 µs (99th percentile 1.696) and 1.67 µs (99th percentile 1.691) for 64 B and 65 B packets respectively. P4DNS latency is higher as we add more parsing logic and match-action stages to this original design. We note that latencies are expected to be significantly better using SDNet releases 2019.1 and the same P4DNS design.

In traditional data centers, DNS requests have inherent switching latency costs. We envision a data center environment where P4DNS sits as a ToR switch transparently resolving DNS queries. P4DNS negates switching latency costs, resolving DNS queries for free by piggy-backing on a packet that is already being switched.

Tail latency is particularly important in data center environments. Fig. 5 shows that P4DNS's latency is far lower than NSD's in all observed cases.

B. Comparison to NSD and Emu

P4DNS represents a midway point in functionality between NSD and Emu. P4DNS is not a fully fledged DNS server, and does not compete with NSD on features. However, as we have seen above, limiting the scope to the most commonly used DNS features [25] enables the design of efficient and high performance hardware DNS caches. Emu took this philosophy to the extreme providing bare-minimum functionality. In Emu, the user must manually select a single domain name for which responses can be issued. P4DNS's match-action pipeline enables a dynamic approach, supporting more entries and more features (e.g., TTL updates) combined with higher throughput. Emu is not fully pipelined, and therefore can not achieve full line rate, unless each stage in the pipeline runs as a separate thread (which is not the case in [2]).

VII. Discussion

P4DNS supports line rate DNS responses for sustained periods of time. P4DNS supports:

- Fast, near-host generation of DNS responses before they traverse the network, supported on all four ports.
- Passive cache updates from DNS responses traversing the network.
- Intelligent cache management strategies.
- Resolution of recursive DNS requests.

Although the P4→NetFPGA workflow enabled many features, we encountered several limitations. First, the generality of the parser was a problem. P4DNS does not require the power of a state machine. However, parsing off the end of a packet results in all parsed bits being invalidated. This restriction means a state is required for each protocol even if the parser state does not subsequently diverge. Instead, if the parser yielded as many valid bits as possible, we could parse the entire packet with one call to `packet.extract` and save a number of states in the parser. The parser is also made complicated as variable length fields are not supported by P4→NetFPGA[3]. Lighter-weight parsing mechanisms enable more complex in-network computing.

Further, we found handling C-style strings cumbersome. This is not entirely surprising, but applications using C-style strings might have to rely heavily on externs.

Finally, although P4DNS' data plane can generate DNS responses at line rate, the control plane has far weaker performance. Control plane scalability depends on host resources, limiting packet processing rate, and leading to drops in the listener function even with a moderate percentage of traffic going to control plane. Application designers should bear in mind that the control plane is unsuitable for high packet rates (which is why in-network computing was originally called for). Therefore applications where critical packets are sent to the control plane will gain little in performance. A direct effect on P4DNS is that P4DNS cannot support TCP-based DNS without significant reliance on externs to manage the state of each connection.

VIII. Related and Future Work

In-network computing has been introduced as a means to accelerate applications and offload hosts [6]. It achieved billions of operations per second for caching applications [26], [27], supporting distributed system functionality [9], stream processing [8] and more. P4DNS can be ported to programmable switches, as it uses no externs, and match their line rate. P4DNS stands out from many of these projects (e.g., [26], [27]) as it integrates an existing protocol rather than design a new one. Further, P4DNS acts as a transparent cache rather than a mandatory part of the system as is the case in Eris [9].

DNS acceleration in hardware was presented in Emu [2]. However, Emu's DNS server supports a single, fixed, A record, and no recursive resolution or dynamic cache updates. Marinos et al. [28] explored accelerating DNS within a traditional software environment and achieved 9× throughput improvements over NSD by specializing the network stack.

In-network computing is gradually being adopted and standardized, with groups such as IRTF COIN [4] looking at the greater picture of realistic deployments and the creation of RFCs. Hopefully, P4DNS will contribute to this discussion.

Future extensions of P4DNS will support any valid length packets and a bigger cache. Matching domain names on a per-label basis is another challenge, requiring multiple chained match-action tables and C-style string parsing, but enabling NS responses with redirection. P4DNS has not implemented security aspects of DNS, such as DNSSEC and DoH, which are beyond the scope of this work. We believe that such separate contributions can be integrated with P4DNS.

IX. Conclusion

We present a P4 implementation of a DNS accelerator capable of running on a NetFPGA. We find latency and

[3]This will be supported in newer SDNet releases

[4]https://datatracker.ietf.org/rg/coinrg/

throughput improvements over state-of-the-art software solutions, by bringing computation closer to the end host. In achievable throughput, P4DNS outperforms NSD by a factor of 52 and Emu by a factor of 10. Our work shows the feasibility for network application caching to be implemented within switches with positive results. We further identify the computational power of the parser as both more than necessary for our needs, and a source of increased resource usage. Importantly, we identify the control plane as a limiting factor in the design of stateful hardware. To enable the scalability of future in-network computingsolutions, designs are required to reduce data plane's dependence on control plane operations.

ACKNOWLEDGMENTS

We would like to thank Gordon Brebner and Stephen Ibanez for their assistance. We acknowledge support by the Leverhulme Trust (ECF-2016-289) and the Isaac Newton Trust.

REFERENCES

[1] Cisco, "Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper," Nov 2018.

[2] N. Sultana, S. Galea, D. Greaves, M. Wójcik, J. Shipton, R. Clegg, L. Mai, P. Bressana, R. Soulé, R. Mortier, P. Costa, P. Pietzuch, J. Crowcroft, A. W. Moore, and N. Zilberman, "Emu: Rapid prototyping of networking services," in *ATC'17*. USENIX, 2017, pp. 459–471.

[3] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward In-Network Computation with an In-Network Cache," in *ASPLOS'17*. ACM, 2017.

[4] D. A. Popescu, N. Zilberman, and A. W. Moore, "Characterizing the impact of network latency on cloud-based applications' performance," University of Cambridge, Tech. Rep. 914, 2017.

[5] Y. Tokusashi, H. Matsutani, and N. Zilberman, "Lake: The power of in-network computing," in *ReConFig'18*. IEEE, Dec 2018, pp. 1–8.

[6] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-Network Computation is a Dumb Idea Whose Time Has Come," in *HotNets'17*. ACM, Nov 2017.

[7] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *NSDI'18*. USENIX, 2018, pp. 51–64.

[8] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, and R. Soulé, "Life in the fast lane: A line-rate linear road," in *SOSR'18*. ACM, 2018.

[9] J. Li, E. Michael, and D. R. K. Ports, "Eris: Coordination-Free Consisten Transactions Using In-Network Concurrency Control," in *SOSP'17*. ACM, 2017.

[10] T. A. Benson, "In-Network Compute: Considered Armed and Dangerous," in *HotOS'19*. New York, NY, USA: ACM, 2019, pp. 216–224.

[11] D. R. K. Ports and J. Nelson, "When Should The Network Be The Computer?" in *HotOS'19*. New York, NY, USA: ACM, 2019, pp. 209–215.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[13] Barefoot Networks, "Tofino," https://barefootnetworks.com/products/brief-tofino/, 2019.

[14] N. McKeown, "Software-defined networking," 2009, infocom Keynote.

[15] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano, "FlowBlaze: Stateful Packet Processing in Hardware," in *NSDI'19)*. Boston, MA: USENIX, 2019, pp. 531–548. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pontarelli

[16] The P4 Language Consortium, "$P4_{16}$ language specification," 2018.

[17] N. Zilberman, Y. Audzevich, A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as Research Commodity," *IEEE Micro*, vol. 34, pp. 32–41, 2014.

[18] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4→NetFPGA Workflow for Line-Rate Packet Processing," in *FPGA'19*. New York, NY, USA: ACM, 2019, pp. 1–9.

[19] P. Mockapetris, "RFC 1034: Domain Names — Concepts and Facilities," IETF, 1987.

[20] ——, "RFC 1035: Domain Names — Implementation and Specification," IETF, 1987.

[21] NLnet Labs, "Name Server Daemon (NSD)," Available at https://nlnetlabs.nl/projects/nsd/about/.

[22] P. Biondi, "Scapy: explore the net with new eyes," *Technical report, EADS Corporate Research Center*, 2005.

[23] Exablaze, "ExaNIC HPT," https://exablaze.com/exanic-hpt.

[24] G. Antichi, M. Shahbaz, Y. Geng, N. Zilbermand, A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski, "OSNT: Open Source Network Tester," *IEEE Network*, September/October 2014.

[25] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, "DNS Performance and the Effectiveness of Caching," *Transactions on Networking*, 2002.

[26] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, K. Changhoon, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *SOSP'17*. ACM, 2017.

[27] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *NSDI'18*. Renton, WA: USENIX, 2018, pp. 35–49. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/jin

[28] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *SIGCOMM '14*. New York, NY, USA: ACM, 2014, pp. 175–186.