Hardware-Accelerated Network Control Planes

Edgar Costa Molero
ETH Zürich
cedgar@ethz.ch

Stefano Vissicchio
University College London
s.vissicchio@cs.ucl.ac.uk

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

ABSTRACT

One design principle of modern network architecture seems to be set in stone: a software-based control plane drives a hardware- or software-based data plane. We argue that it is time to revisit this principle after the advent of programmable switch ASICs which can run complex logic at line rate.

We explore the possibility and benefits of accelerating the control plane by offloading some of its tasks directly to the network hardware. We show that programmable data planes are indeed powerful enough to run key control plane tasks including: failure detection and notification, connectivity retrieval, and even policy-based routing protocols. We implement in P4 a prototype of such a "hardware-accelerated" control plane, and illustrate its benefits in a case study.

Despite such benefits, we acknowledge that offloading tasks to hardware is not a silver bullet. We discuss its tradeoffs and limitations, and outline future research directions towards hardware-software codesign of network control planes.

1 INTRODUCTION

As the "brain" of the network, the control plane is one of its most important assets. Among other things, the control plane is responsible for *sensing* the status of the network (e.g., which links are up or which links are overloaded), *computing* the best paths along which to guide traffic, and *updating* the underlying data plane accordingly. To do so, the control plane is composed of many dynamic and interacting processes (e.g., routing, management and accounting protocols) whose operation must scale to large networks. In contrast, the data plane is "only" responsible for forwarding traffic according to the control plane decisions, albeit as fast as possible.

These fundamental differences lead to very different design philosophies. Given the relative simplicity of the data plane and the "need for speed", it is typically entirely implemented in hardware. That said, software-based implementations of data planes are also commonly found (e.g., OpenVSwitch [30]) together with hybrid software-hardware ones (e.g., CacheFlow [20]). In short, data plane implementations cover the entire implementation spectrum, from pure software to pure hardware. In contrast, there is *much* less diversity in control plane implementations. The sheer complexity of the control plane tasks (e.g., performing routing computations) together with the need to update them relatively frequently (e.g., to support new protocols and features) indeed calls for software-based implementations, with only a few key tasks (e.g., detecting physical failures, activating backup forwarding state) being (sometimes) offloaded to hardware [13, 22].

Yet, we argue that a number of recent developments are creating both the *need* and *opportunity* for rethinking basic design and implementation choices of network control planes.

Need There is a growing need for faster, more scalable, and yet more powerful control planes. Nowadays, even beefed-up and highly-optimized software control planes can only process thousands of (BGP) control plane messages per second [23], and can take *minutes* to converge upon large failures [17, 36]. Parallelizing only marginally helps: for instance, the BGP specification [31] mandates to lock all Adj-RIBs-In before proceeding with the best-path calculation, essentially preventing the parallel execution of best path computations. A concrete risk is that convergence time will keep increasing with the network size and the number of Internet destinations. At the same time, recent research has repeatedly shown the performance benefits of controlling networks with extremely tight control loops, among others to handle congestion (e.g., [7, 21, 29]).

Opportunity Modern reprogrammable switches (e.g., [1]) can perform complex stateful computations on billions of packets per second [19]. Running (pieces of) the control plane at such speeds would lead to almost "instantaneous" convergence, leaving the propagation time of the messages as the primary bottleneck. Besides speed, offloading control plane tasks to hardware would also help by making them traffic-aware. For instance, it enables to update forwarding entries consistently with real-time traffic volumes rather than in a random order.

Research questions Given the opportunity and the need, we argue that it is time to revisit the control plane's design and implementation by considering the problem of offloading parts of it to hardware. This redesign opens the door to multiple research questions including: *Which pieces of the control plane should be offloaded? What are the benefits?* and *How can we overcome the fundamental hardware limitations?* These fundamental limitations come mainly from the very limited instruction set (e.g., no floating point) and the memory available (i.e., around tens of megabytes [19]) of programmable network hardware. We start to answer these questions in this paper and make two contributions.

HotNets-XVII, November 15–16, 2018, Redmond, WA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6120-0/18/11…$15.00
https://doi.org/10.1145/3286062.3286080

First, we illustrate that the next-generation of programmable switches is powerful enough to run many control tasks directly in hardware. Specifically, we implement a working prototype of a hardware-accelerated control plane in P4 [3]. Our approach enables P4-enabled switches' hardware to perform the following tasks, *autonomously* and *at line rate*: *(i)* detect full and gray failures; *(ii)* run distributed path-vector computations that support both shortest-path and BGP-like policies; and *(iii)* update the forwarding state.

Our implementation compensates for the computation and memory limitations with additional packet exchanges. For example, during path computations, each switch only stores the best path, forgetting its alternatives. This implies that more packets have to be exchanged upon configuration or topological changes. Yet, this only induces a marginal cost for hardware implementations, as packet processing takes nanoseconds [32].

Second, we discuss the pros and cons of offloading control plane tasks to hardware. Based on this analysis, we sketch a research agenda centered around the investigation of a software-hardware codesign approach to network control planes, aimed at systematically exploring the tradeoffs of running tasks in software, hardware, or a combination of the two.

Our observations complement recent proposals on hardware offloading for network monitoring tasks [27, 28, 34], congestion control [8], coordination services [18], consensus algorithms [10, 11], and application-level caching [19, 32]. A few proposals, like DDC [25], have also shown how to offload specific functions to the data plane, such as maintaining connectivity. We expand on this intuition, considering any control plane task as a candidate for hardware offloading.

Overall, we think that offloading control plane tasks to hardware has the potential to radically change the way networks are designed in the future.

2 HARDWARE-BASED CONTROL PLANE

Networks are organized around two planes: the control and the data plane. The Control Plane (CP) is the "brain" of the network and is responsible for computing forwarding paths. It can be either logically-centralized, as in SDN networks, or distributed, as in "traditional networks" running distributed protocols (IGP, BGP, etc.). The role of the Data Plane (DP) is simply to forward traffic (as fast as possible) according to the CP decisions. While the DP can be implemented in either hardware or software, the CP is typically implemented in software and involves three main processes:

(1) *Sensing:* The CP monitors the network topology and configuration, in order to detect changes (e.g., link failures) that may require to adapt the forwarding state.
(2) *Notification:* When detecting a change, the CP notifies the path computation component. If the CP is logically centralized [26], the central controller is notified. If the CP is distributed, all the network nodes must be notified about the change.
(3) *Computation:* When becoming aware of a topological change, the routing component of the CP recomputes the forwarding paths. Once new paths are computed, the CP updates the data-plane.

In this section, we show that each step can run directly in hardware, paving up the way for hardware-based CPs.

We use the 4-switches network of Figure 1 as visual support. The figure illustrates how a hardware-based CP senses and retrieves connectivity upon a partial link failure happening between switch B and C.

2.1 Hardware-based sensing

Some forms of hardware-based sensing are already available today. Existing approaches rely on either monitoring properties of the physical medium (e.g., loss of light in an optical fiber) or running the Bidirectional Forwarding Detection (BFD) protocol [22]. BFD sends small echo packets every x ms (50 ms by default [2]) and generates an alert if more than k have not been received.

Challenges Existing hardware sensing schemes can only detect hard failures, such as a link or a node failing, not gray ones. Gray failures are partial failures that affect only a subset of the traffic (e.g., packets matching a specific forwarding entry [16]). The key challenge here is that detecting gray failures requires them to be exerted by actual traffic, preventing simple hardware hello-based mechanisms from working.

Our approach We generalize the concept of BFD to detect both hard and gray failures, in hardware. We program adjacent switches to "acknowledge" the *data-plane traffic* that they exchange, rather than BFD hello packets. While this may seem excessive, acknowledgments only need to contain enough header information for identifying the rule being touched by the packet (e.g., the 32-bits destination prefix). Assuming 32-bits acknowledgements,[1] the overhead would be 267 Mbps for 100 Gbps of traffic with 1500 bytes packet. With minimal size packets (64 bytes), the same volume of traffic would require 6.25 Gbps of acknowledgements.

To avoid acknowledging every single packet, we propose a scheme in which switches synchronously exchange packet counts processed by any given forwarding rule. Specifically, an upstream switch instructs a downstream switch to start and stop counting packets matching a given forwarding rule. When receiving the stop signal, the downstream switch sends the counter back to its upstream which compares it with its own packet count. This process is illustrated in Figure 1 (left), where C, the upstream switch, sends packets to B. The different counter values for the red destination indicate a gray failure which is reported by C network-wide.

[1] Here, we consider the use of protocol-independent switches which do not mandate the use of an Ethernet header.

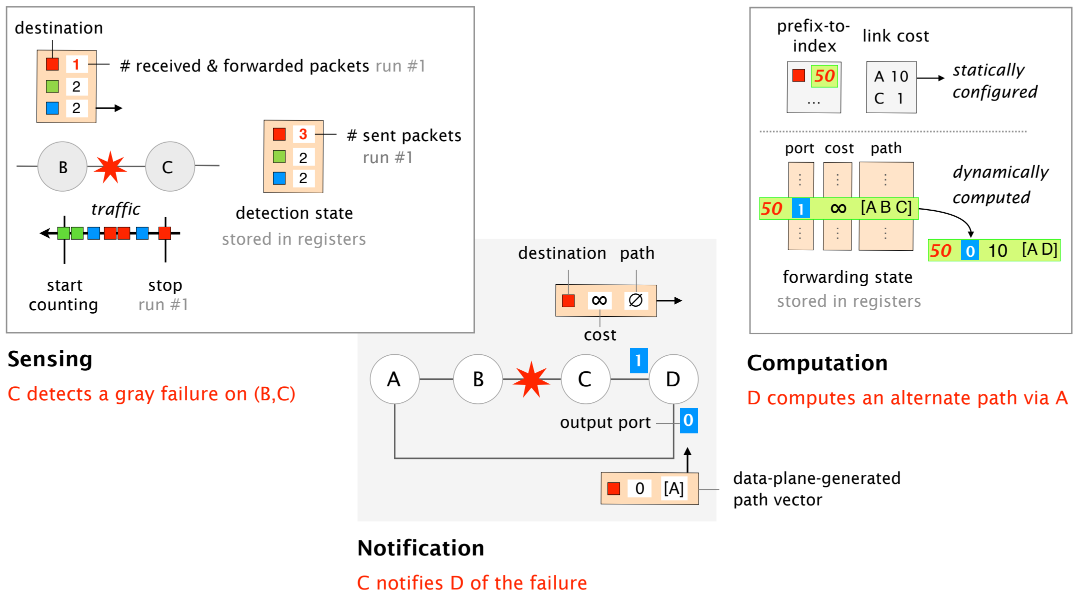

Figure 1: Despite being limited in terms of computation logic and memory, programmable data planes are powerful enough to run key control plane tasks enabling them to compute forwarding state entirely on their own.

2.2 Hardware-based notification

We take inspiration from the simplest, least memory consuming routing protocols, and implement a broadcasting notification mechanism in hardware. As shown in Figure 1, notifications correspond to the generation of path-vector messages. Those messages are also used during the path computation, and carry information on: *(i)* affected destinations; *(ii)* the most updated path (i.e., empty for the link failure in the figure); and *(iii)* its cost (i.e., infinity for failures).

Challenges Broadcasting in hardware poses two main challenges. First, notifications must be exchanged reliably to guarantee correctness. Implementing reliable communication in hardware is challenging as it requires to maintain state, track timers, and deal with the inevitable retransmissions. Second, broadcasting notifications requires extra care to avoid broadcast storms in the presence of physical cycles.

Our approach We deal with packet loss in two ways. First, we classify control packets in high priority queues, reducing the likelihood of packet loss. Second, we leverage that the cost of processing a packet is almost negligible in hardware, and either duplicate messages k times, for notifications, or repeat them regularly, e.g. every few ms, for regular state exchange. While continuously repeating state exchange guarantees its eventual consistency network-wide, there is still a small probability that some switches will not receive any of the k retransmitted notifications, leading to a partially converged network. In future work, we intend to develop a lightweight form of reliable message exchanges.

To avoid broadcast storms, the originator switch attaches its identifier and a sequence number to the broadcasted packet. Each switch maintains a register with the last sequence number observed for every other switch. Whenever a switch receives a broadcast message, it checks whether the sequence number is smaller than the one stored for the message originator, and drops the packet if it is the case. The sequence number is increased by one during the next broadcasting.

2.3 Hardware-based computation

We implement a distributed path vector routing algorithm in hardware in which switches exchange vectors and locally select the vector with the best attributes, e.g. the one with the lowest cost or the one with the highest preference. By doing so, our hardware computation supports policy-based (i.e., BGP-like) routing logic.

Challenges A key challenge is that the computation logic available is limited and geared towards forwarding pipelines, not distributed algorithms. For instance, P4 does not support basic constructs like loops. On top of that, resources are heavily limited (in terms of size and data structure types), which clashes with the typical choice of routing protocols to maintain *a lot* of state, such as all the routes received or the entire network map. Finally, supporting routing policies adds an extra level of complexity as the presence of policies render many routing problems computationally-hard [15].

Our approach To manage complexity and reduce the amount of state maintained by each switch, we only make them store the best path and its attributes. This simplifies the computation as it removes the need to iterate: a switch only needs to compare the received attributes with the currently best known path, and possibly adapt the latter accordingly. Of course, it also reduces the amount of state maintained by each switch to the bare minimum.

Observe that this strategy is sufficient to compute a new best path if some input changes, provided each switch re-advertises its best known path upon a change. To ensure this, the failure notifications are flooded and necessarily trigger a re-advertisement. While doing so leads to more messages than software CPs storing alternative paths, we stress that this is not a problem since hardware-based computation can process *billions* of such packets per second [19].

Finally, we leverage the seminal results from Sobrinho [35] to compute the outcome of policy-based protocols such as BGP in hardware. Those results show that *generic* path vector protocols can emulate the semantics of policy-based protocols, if the right set of costs is chosen. This observation enables to move the complexity of dealing with policies from the protocol to the path costs. We show how our hardware-based CP encodes typical BGP policies (prefer customer over peer over provider routes) in Section 3.

3 PRELIMINARY IMPLEMENTATION

We now describe a preliminary P4 implementation of our hardware-accelerated control plane and illustrate its usefulness through a case study in which switches converge entirely on their own, for both intra- and inter-domain destinations.

3.1 Implementation

We implement our algorithms in P4$_{16}$ [5] and use the bmv2 [6] behavioral model to test them. We also implement a software-based control plane logic which is in charge of populating the switches initial state. Overall, our implementation consists of 1800 lines of P4 and 3000 lines of Python code. We performed our experiments on a server equipped with a 2×12 Xenon E5-2670 2.30GHz, 128GB RAM and running Ubuntu 16.04. In the future, we intend to adapt and run our algorithms on Tofino switches [1] with a dynamic control plane.

Challenges We enumerate some of the implementation-related challenges we encountered and how we solved them.

- *Modifying the forwarding state at line rate:* In P4, the content of the forwarding tables is provisioned by the control plane through dedicated APIs. Unfortunately, the content of these match-action tables cannot be modified at line rate unless the hardware architecture supports it. We solved this challenge by making the LPM match-action tables (TCAM) point to stateful objects (i.e., registers implemented using SRAM), which can then be modified at line rate.
- *Loops:* P4 does not allow loop constructs. We address this by unrolling loops and performing each iteration step in parallel. If the loop is longer than the maximum number of parallel steps supported by the switch, we recirculate the packet.
- *Generating packets:* P4 does not enable to instruct the switch to generate packets. To address this limitation, we use the actual traffic as a carrier for our protocols. If not enough traffic is present, we periodically send empty packets from the network edge.
- *Parsing limits:* Current hardware switches can parse up to 300 bytes per packet to maintain line rate [33], hence limiting the amount of data switches can exchange in a packet. We use this limit as constraint in our design, mandating the switches to generate smaller packets.

3.2 Intra/inter-domain routing ... in hardware!

We now describe the key insights behind our path-vector implementation and how it manages to compute intra-domain and inter-domain paths.

Computing intra-domain routes Each switch keeps the best cost, path and output port towards every other switch in stateful registers (see Figure 1). Switches periodically advertise a vector $[(ID_i, cost_i, path)_i, ...]$. To generate it, the switch reads a fixed amount of register entries and pushes them into a new header. If the number of switches is bigger than the maximum number of times we can read a register, we recirculate the packet. Once the entire vector is placed into the packet, the switch sends it to all its neighbors.

Upon receiving a vector, a switch parses a fixed amount of fields and runs the shortest path computation in parallel for all of them. Specifically, the switch checks if the cost stored plus the cost to reach the advertising neighbor is smaller than the advertised cost. To avoid count-to-infinity, the switch also verifies that it is not in the path. If both hold, the switch updates its register with the new cost, path and output port. This process is repeated until the entire vector is processed. If any cost was changed during the updating phase, the switch generates an advertisement.

Our prototype assumes that the switch receives a link down notification. Upon receiving it, the switch iterates through its distance vector register and forgets all the routes using that link. Finally it generates an advertisement.

Computing inter-domain routes To compute new egresses for inter-domain routes, switches keep both: *(i)* a register that maps a prefix to the best exit point known in the network; and *(ii)* a register with prefixes that the switch can reach from its external peers. To support the normal BGP decision process, switches also keep the AS path length for each route along with the type of peering relationship (e.g., customer/peer/provider) for all the egress points.

The computation process is triggered once a switch receives a prefix withdrawal from one of its peers. It then proceeds in two steps. First, it broadcasts a special packet indicating that the prefix cannot be reached though that egress. Second, the switch removes the corresponding route (if it exists).

Upon receiving a broadcasted withdrawal, a switch looks at its registers to check whether it affects its egress. If so, it removes the route. If it uses another egress or if it knows how to reach the prefix via one of its direct peers, the switch broadcasts a message announcing the backup egress.

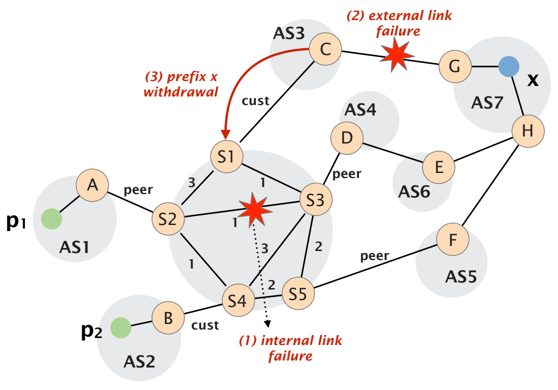

Figure 2: Case study topology

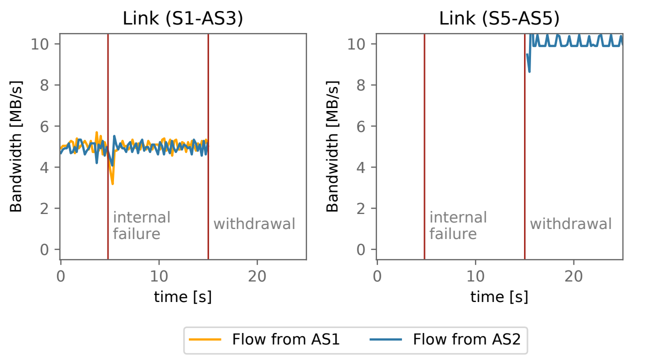

Figure 3: Per flow bandwidth at two egress points towards prefix X. Red vertical lines indicate network events

Each switch runs a BGP-like route selection algorithm upon receiving a backup announcement and compares the best route they currently know with the advertised one. Route selection is done as follows: if the local preference is higher, the egress is accepted as a backup; if the local preference is equal, the egress with shortest AS path length is selected; if the path lengths are equal, the shortest distance to the egress is used; otherwise, the route is rejected. Besides computing the best egress point, upon an egress update, switches also immediately block all the traffic that violates export policies (e.g., traffic from a peer to a peer).

3.3 Case study

We now show that our implementation enables programmable switches to converge on their own upon different failures.

Methodology: We use a small topology consisting of 5 internal switches running our hardware-based control plane algorithms (Figure 2). Each switch is connected externally to either one customer or one peer.

We generate two TCP flows, one from AS1 and one from AS2, both flows have network X (in AS7) as a destination. To show that switches can react autonomously to internal and external failures, we introduce two events at different times. First, we fail the internal link S2-S3, which will trigger the intra-domain computation. Then, after some seconds, we send a withdrawal for prefix X to S1 from AS3, henceforth triggering the inter-domain computation enabling the switches to find the second best egress for destination X.

We start the experiment with a converged network in which the control plane has populated the forwarding register that maps external prefixes to best egress IDs using BGP. Each switch also stores in memory which external prefixes can be reached via itself. To avoid being CPU bounded during the study, we set the bandwidth of every link to 10Mbps.

Results We study how and for how long failures affect traffic that crosses our hardware-based control plane network. Figure 3 depicts the throughput observed over the link S1-AS3 and S5-AS5. Initially, we see that both flows are using S1-AS3 to leave the network (i.e., using the customer link) and, as such, get on average a throughput of 5Mbps.

We first fail the link S2-S3 and send a notification to the affected switches 200ms after the failure, which triggers the intra-domain routing algorithm. As we can see in Figure 3 (left), the failure affects both flows for a short period of time, mainly due to the detection delay.

We then fail the link S1-AS3 by sending a withdrawal to S1. S1 immediately removes its route and starts dropping packets.[2] S1 then broadcasts that network X cannot be reached, making S3 and S5 broadcast back their alternative egress point. This in turn triggers the inter-domain route selection algorithm on all switches. Since S3 and S5 have the same local preference, the tie is broken using the AS path length making S5 the preferred egress. As S5-AS5 is a peer link, only the customer flow from AS2 is allowed (due to BGP export policy violations). Accordingly, we can see in Figure 3 (right) that S1 stops forwarding traffic and that the flow coming from AS2 starts egressing at S5-AS5 at 10Mbps.

Overall, we see that our data-plane implementation is able to automatically converge while respecting the BGP policies.

4 HARDWARE IS NOT "ALL ROSES"

In this section we discuss the pros and cons of offloading control plane tasks to hardware.

The pros A key motivation to offload control plane tasks to programmable hardware is that most control plane operations are compatible with programmable hardware's capabilities. In our approach, for example, sensing, notification and computation are implemented by exchanging packets of a given format, processing them in a predefined way, updating the hardware state, and generating packets of potentially a different format as a result. Receiving, elaborating and generating packets is exactly what the hardware is powerful at.

In addition, it is very natural for the hardware implementation of control plane tasks to be driven by data-plane traffic, so that the forwarding state is computed and updated according to the actual data traffic. In our prototype, forwarding

[2]We leave for future work the implementation of a mechanism to maintain connectivity while learning the backup egress.

entries tend to be updated in an order consistent with per-destination traffic volumes: since packets trigger actions from the hardware-based control plane, traffic for destinations carrying more traffic are probabilistically rerouted first. This produces less packet losses than updating forwarding entries in a random order, as software control planes often do.

Even better, running the control plane in hardware unlocks capabilities that *cannot* be easily implemented otherwise, such as the cheap and prompt detection of gray failures (Section 2). In fact, state-of-the-art approaches to detect gray failures either generate and post-process a huge amount of data-plane traffic, like [16], or do use programmable hardware to track packets as they cross different devices [24].

Maintaining connectivity *during* failures is another case where hardware offloading is strictly needed as waiting for the control plane to react would necessarily lead to packet losses. This is the reason why existing fast-reroute frameworks, like [13], pre-load backup paths in the switches, so as to activate them, in hardware, as soon as the failure is detected. Of course, pre-loading backup states consumes a lot of memory and is generally not scalable with respect to the exponential number of possible failure cases. Recent works, like DDC [25], show that performing control-plane computations in hardware enables to break this otherwise-fundamental tradeoff between switch memory and reaction time.

Finally, being able to take forwarding decisions entirely in the data plane, without any control plane or controller, can be critical in environments where microseconds matter. For example, in data center networks where traffic loads change rapidly, decisions have to be taken almost instantaneously. Having a control-loop that goes though a software control plane leads to outdated decisions. Recent research, has shown that being able to load-balance traffic entirely in the data plane is not only possible, but surprisingly simple and effective (e.g., [7, 14, 21]).

In general, the investigation of additional use cases opened by the hardware implementation of control plane capabilities is an interesting direction for future research.

The cons Hardware offloading is not infinitely expressive: some tasks *cannot* be delegated to hardware. For example, hardware sensing cannot be used for detecting software failures, hence detection and reaction mechanisms to these types of failures must remain in software.

Also, even when technically possible, offloading tasks to hardware might not be desirable. For example, it makes little sense to implement protocols like BGP and the underlying TCP in hardware. First, a hardware implementation would consume many hardware resources for little or no gain—especially if we consider that BGP performance is often limited by the TCP's internal algorithms [4]. Second, performance and capabilities cannot be radically changed without revisiting the implementation of the protocol on multiple administrative authorities.

For the remaining control plane tasks for which offloading to hardware can come with benefits, a major limitation is represented by the scalability of hardware implementations, a characteristic for which a software component of the control plane is likely to be needed in many realistic settings. In particular, hardware offloading is likely to scale poorly with the number of control plane tasks. On the one hand, hardware resources, like ASICS registers or memory, are typically scarce, and hard (and expensive) to scale. On the other hand, offloading control plane tasks are likely to consume a lot of hardware resources, e.g., because of the need to store messages, data, and computation parameters in hardware. Combined together, these two factors create the need for limiting the number of tasks offloaded to hardware, and hence to accurately select which functions to offload to hardware.

Carefully, and perhaps dynamically, allocating resources to different hardware computations is an interesting challenge to address in future research.

5 HARDWARE-SOFTWARE CODESIGN MEETS CONTROL PLANES

So far we have shown the benefits but also the limitations of offloading task to hardware. This duality indicates that *accelerating* the control plane by offloading *some* tasks to hardware and keeping others in software can lead to control plane design points of great practical interest.

Our vision is that the search for an optimal design point can be formalized as a hardware-software codesign problem and solved using the classical 4-phases methodology [12]: specification, analysis, synthesis and validation. Instantiating this methodology to our context is a challenging problem which calls for interesting future research contributions.

More specifically, the specification phase requires precise models of the current control plane functions (e.g., failure detection, routing, updates). These models should allow for the efficient evaluation of the performance and cost of performing each function in software, hardware, or a mix of both. Interestingly, realistic models and cost functions must take into account the dynamic interaction between distinct control plane components, which potentially make the cost of each specific design higher than the cost of running each component separately. Furthermore, these models should also account for the cost of "hybridizing" control plane tasks by allocating some parts in software and others in hardware (e.g., accounting for the cost of synchronizing both entities).

Likewise, the analysis and synthesis phases call for the design of efficient search heuristics which leverage domain-specific knowledge to navigate the exponential space of possible hardware-software codesigns (a problem known to be NP-hard [9]). In particular, we plan to explore if it is possible to learn probabilistic models of the likelihood that a particular design is better than another.

Finally, the validation of (partially) offloaded control planes opens up interesting verification questions such as how to ensure that a specific design will perform accordingly both feature- *and* performance-wise.

REFERENCES

[1] Barefoot Tofino. https://barefootnetworks.com/products/product-brief-tofino/.

[2] Cisco IOS. IP Routing: BFD Configuration Guide. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bfd/configuration/15-mt/irb-15-mt-book/irb-bi-fwd-det.html.

[3] The P4_16 language specification - version 1.0.0. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html.

[4] TCP Behavior of BGP, 2012. https://archive.psg.com/121009.nag-bgp-tcp.pdf.

[5] P4-16 Language Specification., 2018. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[6] P4 behavioral model., 2018. https://github.com/p4lang/behavioral-model.

[7] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 503–514, New York, NY, USA, 2014. ACM.

[8] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker. Hotcocoa: Hardware congestion control abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 108–114, 2017.

[9] P. Arató, Z. Á. Mann, and A. Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 10(1):136–156, 2005.

[10] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review*, 46(2):18–24, 2016.

[11] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 5, 2015.

[12] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[13] C. Filsfils, P. Mohapatra, J. Bettink, P. Dharwadkar, P. D. Vriendt, Y. Tsier, V. V. D. Schrieck, O. Bonaventure, and P. Francois. BGP Prefix Independent Convergence (PIC) Technical Report. Technical report, Cisco, 2011. http://www.cisco.com/en/US/prod/collateral/routers/ps5763/bgp_pic_technical_report.pdf.

[14] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 225–238, New York, NY, USA, 2017. ACM.

[15] T. G. Griffin and G. Wilfong. An analysis of bgp convergence properties. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 277–288, New York, NY, USA, 1999. ACM.

[16] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 139–152. ACM, 2015.

[17] T. Holterbach, S. Vissicchio, A. Dainotti, and L. Vanbever. SWIFT: Predictive fast reroute. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 460–473. ACM, 2017.

[18] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[19] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136. ACM, 2017.

[20] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite cacheflow in software-defined networks. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 175–180. ACM, 2014.

[21] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, page 10. ACM, 2016.

[22] D. Katz and D. Ward. Bidirectional Forwarding Detection. RFC 5880, 2010.

[23] A. Lambrianidis and E. Nguyen Dyu. Route server implementations performance. Euro-IX Forum, Amsterdam, The Netherlands, 2012.

[24] Y. Li, R. Miao, C. Kim, and M. Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 481–495. ACM, 2016.

[25] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. Ensuring connectivity via data plane mechanisms. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 113–126, Lombard, IL, 2013. USENIX.

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

[27] S. Narayana, A. Sivaraman, V. Nathan, M. Alizadeh, D. Walker, J. Rexford, V. Jeyakumar, and C. Kim. Hardware-software co-design for network performance measurement. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 190–196. ACM, 2016.

[28] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98. ACM, 2017.

[29] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review*, 44(4):307–318, 2015.

[30] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The design and implementation of open vswitch. In *NSDI*, volume 15, pages 117–130, 2015.

[31] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), Jan. 2006.

[32] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA, 2017. ACM.

[33] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 150–156, New York, NY, USA, 2017. ACM.

[34] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176. ACM, 2017.

[35] J. L. Sobrinho. Network routing with path vector protocols: Theory and applications. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 49–60. ACM, 2003.

[36] A. Taylor, B. Rudolph, D. Spierling, and J. Moos. An ixp route server test framework. Euro-IX Forum, Barcelona, Spain, 2017.