Tofino + P4: A Strong Compound for AQM on High-Speed Networks?

Ike Kunze, Moritz Gunz, David Saam, Klaus Wehrle, Jan Rüth
RWTH Aachen University, *Chair of Communication and Distributed Systems*
{kunze, wehrle, rueth}@comsys.rwth-aachen.de, {moritz.gunz, david.saam}@rwth-aachen.de

***Abstract*—Bufferbloat and congestion in the Internet call for the application of AQM wherever possible: on backbone routers, on data center switches, and on home gateways. While it is easy to deploy on software switches, implementing and deploying RFC-standardized AQM algorithms on programmable, pipeline-based ASICs is challenging as architectural constraints of these ASICs were unknown at the time of standardization. In this work, we call for reigniting the work on AQM algorithms by illustrating the difficulties when implementing the PIE AQM in three fashions on an Intel Tofino switching ASIC. All our implementations come with trade-offs, which, in turn, have a significant impact on their performance. The conceptual challenges further suggest that it is currently not possible to implement a fully RFC-compliant PIE version on Tofino. We find that it is non-trivial to transfer RFC recommendations to the resource-constrained Tofino, operating at hundreds of gigabit per second. We thus argue that there is a need for AQM specifications that acknowledge the omnipresence of congestion and include architectural constraints of programmable ASICs into their design.**

***Index Terms*—AQM, P4**

I. Introduction

Congestion in the Internet can appear in various places: at low bandwidth last-mile links [1], at overloaded backbone links [2], or within data center networks [3]. Together with the increased awareness of bufferbloat [4], this observation has led to an intensified research on Active Queue Management (AQM) in the past years. Proposals, such as CoDel [5], PIE [6], and CAKE [7], show that congestion and the induced delays can be effectively avoided on last-mile links when giving congestion-controlled end-hosts timely feedback.

Even though these AQM algorithms can be implemented rather easily on software-based last-mile gateways [8] and deployed via software updates, backbone and data center links are typically equipped with fixed-function high-speed switching ASICs. While superior in speed, they only offer a limited set of functions, often lacking AQM algorithms or only offering variants of RED [9] that are known to be notoriously hard to parameterize correctly [10]. Aggravatingly, fundamental changes to the ASIC (such as a new AQM implementation) have traditionally required building a new ASIC involving costly hardware updates for network operators.

The P4 [11] programming language and the associated re-programmability of modern switching ASICs promise a novel way out. Although arguably not designed for managing buffers and packet scheduling, P4 allows defining the forwarding behavior and packet processing strategies on the data plane. As of today, this is the best chance there is for implementing AQM functionality without having to redesign the hardware.

978-3-903176-32-4

Research has thus tried to design and implement P4-enabled AQM algorithms. Even though many algorithms are designed for simplicity and are expressible in P4, most works (e.g., [12]–[14]) ignore the real-world constraints put in place by today's match-action pipelines, such as hard limits on multiplications and memory accesses [15]–[17] or access to the queue state. Thus, they conceptually fail to run on modern high-speed switching ASICs. Consequently, congestion management at backbone or data center links, even if they were to utilize reprogrammable ASICs, remains a challenge.

In this paper, we contribute to AQM development by showing how key characteristics of AQM algorithms contradict hardware constraints of modern ASICs at the example of the Intel Tofino and portraying how AQM algorithms can still be implemented or at least approximated using the existing capabilities. To this end, we find that the PIE AQM can be approximated, and we implement three flavors using P4 and evaluate their performance on an Intel Tofino ASIC, hereby pointing out limitations, advantages, and design paths of the different approaches. Specifically, this work contributes the following:

- We analyze different AQM algorithms and show their mismatch to the capabilities of the Intel Tofino.
- Using $P4_{16}$ and Intel SDE 9.2.0, we implement three PIE variants on a Tofino switch and show why inherent trade-offs can make them fail to control the buffer. Our implementations are available at [18].
- We argue that future AQM designs should incorporate hardware constraints of switching ASICs, e.g., missing information backflow and limited memory access patterns and calculations, to make them applicable in the Internet core and within data center networks.

Structure. Sec. II first presents different AQM algorithms and how they have been implemented in related works. We then introduce the operating principles of P4-programmable ASICs in Sec. III, before we analyze the mismatch between AQM design and the capabilities of the Intel Tofino switch. Sec. V then discusses possible ways of bringing AQM to Tofino, some of which we demonstrate in Sec. VI by discussing the challenges in implementing three variants of the PIE AQM. In Sec. VII, we then investigate how the ASIC-dictated design space impacts the performance of the three PIE variants. Lastly, we discuss and summarize the impact of our findings on AQM and ASIC design in Sec. VIII and conclude this paper.

II. Active Queue Management and its Implementation in Related Works

Active Queue Management (AQM) algorithms are an essential countermeasure to bufferbloat [4] and designed to maintain a high level of throughput while keeping delays low. In essence, they first detect network congestion, e.g., based on the *queue length*, and subsequently give early feedback to the end-hosts, e.g., in the form of dropping packets. For the remainder, we use *drop* synonymously for all forms of giving feedback, including marking using Explicit Congestion Notification (ECN) and more fine-grained mechanisms [19], [20].

There are numerous AQM proposals, all varying in how they detect congestion and in how and when they give feedback. In the following, we present three popular AQM algorithms.

Random Early Detection (RED). RED [9] associates a large average *queue length* with congestion and notifies end-hosts by randomly dropping packets based on a *drop probability*. This probability is proportional to and calculated based on an exponential weighted moving average (EWMA) of the queue size. If the average is below a minimum threshold, RED does not drop packets to allow for transient bursts; if a maximum threshold is exceeded, all packets are dropped. In between, the probability of dropping an incoming packet increases linearly with the queue size from 0 to a maximum drop probability.

Controlled Delay (CoDel). CoDel's [21] design keeps the *queuing delay*, measured as the time a packet has spent in the buffer, around a constant target value. If the minimum occurring delay is greater than the target for a prolonged time (*interval*), CoDel starts dropping packets. Then, it schedules further drops in ever shorter succession using $t_{next} = t + \frac{interval}{\sqrt{n_{drops}}}$ until the observed delay eventually falls below target.

Proportional Integral Controller Enhanced (PIE). PIE [6], [22] uses an eponymous proportional-integral controller that periodically samples the *queuing delay* to update its *drop probability* (*drop$_{prob}$*). The update $p = \alpha \cdot (delay_{ref} - delay) + \beta \cdot (delay_{old} - delay)$ bases on the difference of the current delay to the previous delay (*delay$_{old}$*) and a reference delay (*delay$_{ref}$*), scaled using system-specific parameters α and β as illustrated in Fig. 1 (a). The update p is additionally scaled based on the current value of *drop$_{prob}$* to make PIE less aggressive for low congestion levels (Fig. 1 (b)). After updating *drop$_{prob}$* (Fig. 1 (c)), *drop$_{prob}$* is exponentially decreased when no delay is noticeable anymore (Fig. 1 (d)).

The control parameters *delay$_{ref}$*, the sampling rate (*t$_{Update}$*), as well as α and β need tuning to the local conditions. RFC 8033 [6] provides guidelines for adjusting α and β subject to changes of *delay$_{ref}$* and *t$_{Update}$*. If *delay$_{ref}$* is reduced, α and β should be increased by the same order of magnitude; for each halving of *t$_{Update}$*, α should also be halved while β should be increased by $\alpha/4$. Finally, PIE should sample at least 2-3 times per round-trip time (RTT). In this work, we refer to the RFC version of PIE [6].

AQM implementations in P4. The presented algorithms have already been implemented using the P4 programming language [11]. Laki et al. [13] design an AQM testbed for which they implement both RED and PIE. While they implement PIE without issues, they, however, need to approximate RED's drop probability using table lookups of the queue lengths as it cannot be computed directly. Similarly, Kundel et al. [12] implement CoDel using a lookup table to approximate the square root function needed to compute the next drop time.

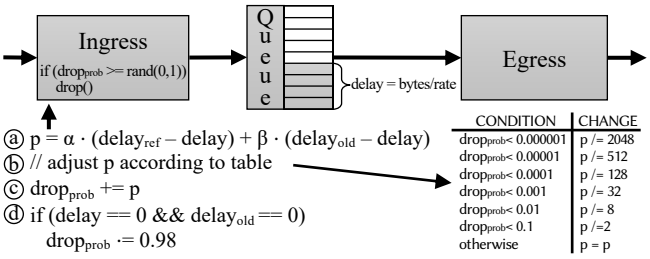

Fig. 1. PIE calculates a drop probability based on the latency within the queue. Based on the drop probability, PIE then randomly drops packets at enqueuing.

Even though these groups implement state-of-the-art AQM algorithms in P4, their implementations only target P4's behavioral model. While this software switch implements all standardized P4 functions, it has no strict resource limitations. Thus, it stands to argue whether these works on AQM in P4 are directly transferable to real switching ASICs as these come with specific constraints, which we discuss next.

III. Constraints of Switching ASICs

ASIC-powered switches are becoming increasingly customizable with P4 which allows defining their forwarding behavior. Commercially available P4-enabled switching ASICs, such as the Intel Tofino [23], offer terabits of performance. Yet, to achieve their high performance, they restrict operations to those that can be executed quickly and with predictable delay.

Pipeline Operation Mode. The imposed limitations stem from the ASICs' pipeline operation mode illustrated in Fig. 2. It consists of a programmable packet parser, an ingress with several stages of match-action units (MAUs), an ingress deparser, a buffer, and a traffic manager followed by a similarly structured egress, again with several stages of MAUs. Packets are first parsed in the parser before they are handed over to the ingress, where one decides the forwarding behavior and selects an egress queue. The traffic manager then enqueues the packet accordingly (this is the queue that can become congested) before the egress pipeline can apply further processing to the packet. Note that the traffic manager itself is not programmable using P4, but where AQM algorithms would typically reside.

Packet Processing. All desired processing after parsing is performed with the help of MAUs. Each unit has matching logic using SRAM or TCAM to perform lookups based on a key, e.g., looking up a next-hop based on a destination address. The action part consists of a set of ALUs that can perform arithmetic and logical operations on data, e.g., coming from packet headers, or stateful elements, such as registers or counters. Conceptually, this processing is not as flexible as on a software switch, as there are hard limits on the number

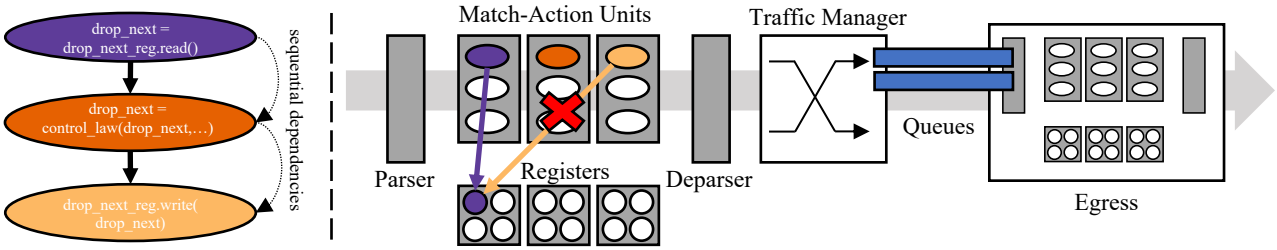

Fig. 2. The pipeline of an ASIC-based switch comprises programmable packet parsers and de-parsers, an ingress with several stages of MAUs, a buffer, a traffic manager, and a similarly structured egress.

and types of operations [15]–[17]. For example, $P4_{16}$ allows multiplications of unsigned integers; we can easily perform many of these multiplications on the behavioral model. Most real ASICs, however, lack any multiplication support as it is a time-intensive operation rarely needed for packet header manipulations. On Tofino, we can compute a limited number of multiplications involving powers of two, but the multiplicand must be statically defined (i.e., from a table lookup or as a compile-time constant).

Placement. All (not parser-related) operations that are expressed in P4, be it a simple if()-statement or any calculation, are mapped to the finite number of MAUs. Further, each packet can only pass through an MAU once (without recirculating the packet through the switch), as each algorithm must complete within a certain time. It is the platform-specific P4-compiler's job to first synthesize the P4 code to a set of MAUs and subsequently find a suitable layout (i.e., a directional graph) that can be mapped onto the switch's resources. For example, different assignments or independent lookups can be parallelized while dependent matches (i.e., the output of a match is the input of another match) need to happen sequentially. In reality, this means: Even if the programmer wrote valid P4 target-specific code, it might still happen that the resources cannot be adequately allocated.

For an illustration, consider the left side of Fig. 2, which shows a naïve implementation of parts of CoDel. The first operation (violet oval) reads out the value of drop_next_reg and is placed in the first MAU. The second (orange) and third (yellow) operations depend on the respective previous operations and can thus only be placed in subsequent MAUs.

Memory Access. The placement problem becomes more complex when using memory, such as tables and registers. Following the reconfigurable match table (RMT) principle [24], memory is stage-local, i.e., statically assigned to one MAU each, and can thus only be accessed once per pipeline pass (*single access rule*). The access to stateful elements, such as registers, further has to be atomic and is mostly limited to one simple read-update-write operation. On Tofino, more expressive update functionality in the form of a small microprogram is possible. This feature is called a *register extern*.

Again looking at Fig. 2, the violet and yellow operations placed in the previous paragraph both require access to drop_next_reg. Yet, reading drop_next_reg in the first MAU statically assigns the memory to that MAU. Consequently, the yellow operation cannot access drop_next_reg from the third MAU, and there is thus no suitable resource allocation.

Use of Externs. These conceptual constraints, as well as the limited physical resources, render it practically difficult or infeasible to run code targeting the behavioral model on a real ASIC. Looking back at the related works, we found that they are either not portable to Tofino or require significant modifications to synthesize a working implementation, in parts heavily relying on special Tofino externs (i.e., custom functions not specified in P4). Yet, AQM designs should not build upon such specialized functionality but rather on a *most common subset of functionality*. In this context, we are interested in which characteristics of these AQM algorithms specifically do not map to the hardware properties of pipeline-based ASICs that do not have the support of special externs.

IV. Bringing AQM Algorithms to Tofino

With respect to their implementability on Tofino, AQM algorithms are characterized by more than the high-level features presented in Sec. II. In addition to the congestion metric and the drop-mechanism, the flow of information, as well as the memory access patterns, are especially critical due to the strict constraints imposed by the pipeline structure.

In the following, we analyze the *three* previously presented AQM algorithms regarding these *four aspects* and summarize our findings in Table I. We roughly rate the aspects regarding their implementability as feasible (✓), needing workarounds (●), and impossible/infeasible (✗). Accounting for our observation regarding externs, we put special emphasis on differences between using a full-fledged Tofino ($Tofino_{full}$, ■) and a Tofino where we restrict the use of externs to basic register functionality ($Tofino_{reg\text{-}only}$, ■), which we distinguish by color. Note that the selected aspects only serve as examples, and there might be even more impacting the implementability.

Congestion Metric. Detecting congestion mostly bases on the queue length [9], [10] or the queuing delay [5], [6]. The queue length is directly available on Tofino, and the queuing delay can be derived with little additional effort. However, RED uses an EWMA of the queue length, which is not feasible using regular ASIC arithmetics (✗). Yet, $Tofino_{full}$ has a special extern for the required calculations (✓). Note that there is also work on approximating the EWMA using P4 [25], but only on the behavioral model. In contrast, the packet delay used by CoDel and PIE is doable (✓/✓).

Congestion Reaction. Most AQM algorithms give feedback by randomly dropping packets based on a drop probability.

	RED	CoDel	PIE
Congestion Metric	✗/✓	✓/✓	✓/✓
Congestion Detection/Reaction	●/✓	●/✓	●/●
Information Backflow	✗/✗	✓/✓	✗/✗
Memory Access	✓/✓	✗/✓	✗/✗

TABLE I
KEY PRIMITIVES OF AQM ALGORITHMS AND THEIR IMPLEMENTABILITY ON A FULL-FLEDGED (■)/ RESTRICTED (■) TOFINO. LEGEND: FEASIBLE (✓), NEEDS WORKAROUNDS (●), AND IMPOSSIBLE/INFEASIBLE (✗).

The random drop itself is available on Tofino, while there exist different techniques for calculating the probability. RED calculates the probability in proportion to its congestion metric; this is not possible on $Tofino_{reg\text{-}only}$. However, as the thresholds are chosen statically, and as the calculation relies on momentary information, it can be reasonably approximated using table lookups (●). $Tofino_{full}$, in turn, provides a dedicated extern for this functionality (✓). The same is true for the square root operation used in CoDel, as it can be approximated for $Tofino_{reg\text{-}only}$ and has an extern on $Tofino_{full}$ (●/✓). In contrast, PIE's update formula is too complex for direct calculation on both platforms, yet it can also be approximated (●/●).

Information Backflow and Drop Position. Congestion detection requires timely information on the queue state, and the decision to drop a packet must be known when actually dropping the packets. On Tofino, queue information is only available in the egress when a packet has been dequeued; it further represents the queue state at the time of dequeue. This means that only AQM algorithms that can be fully executed after dequeue are supported. CoDel conforms to this working principle (✓/✓). RED and PIE, on the other hand, work on incoming packets and thus require queue state information at packet arrival before the packet is actually enqueued. Such functionality is currently not supported on Tofino (✗/✗). The only possible yet infeasible option is to use the dequeue information and feed it back to the ingress.

Memory Access Patterns. Most AQM algorithms utilize state variables for their operation, but access to the corresponding register memory on Tofino is generally constrained. PIE requires two distinct accesses to the drop probability: once to scale its update value and once to actually update the probability. While entirely impossible without externs, the involved calculations are also too complex for Tofino *register actions* (✗/✗). Similarly, CoDel accesses the next drop time twice: first, to check whether it has already passed, and second, to compute a new one if the old one has expired. This is too complex for $Tofino_{reg\text{-}only}$, but the required operations can be expressed using Tofino *register actions* (✓). RED, on the other hand, does not have such issues (✓/✓).

Takeaway. *The three analyzed AQM algorithms require several key primitives that are not natively supported by the pipeline-based Tofino as inherent hardware properties are not considered in AQM RFCs. In addition to having timely information on the queue state in the ingress, the stage-locality of stateful elements, combined with the limits on the number of operations that can be executed per stage, are the major challenges that currently make an RFC-compliant implementation of AQM algorithms on ASICs hard if not outright impossible. Dedicated externs make life easier, but they are currently available on few platforms, and AQM design should rather use commonly available primitives.* What is consequently needed are AQM algorithms that respect general hardware constraints of high-speed switching pipelines without too much relying on special extern support. In an effort to illustrate possible ways forward, we next explore different paths of how to tackle this problem.

V. RESEARCH OBJECTIVES

With the current state of P4 and ASICs, innovating on AQM algorithms or freely designing new functionality is only possible outside of the traffic manager, e.g., by programming the ingress or egress pipelines. This, however, pushes P4 beyond its intended purpose. Furthermore, current AQM algorithms do not consider the aforementioned hardware constraints of high-speed ASICs, which, together, presents a major roadblock in bringing state-of-the-art AQM functionality to high-speed links. Yet, there are at least two ways to move forward: (a) approximate existing AQM algorithms so that they fit on ASICs while preserving their core behavior, and (b) design new algorithms that account for hardware properties of ASICs.

Design Space for AQM on ASICs. Falling in the latter category, Zhang et al. [17] propose ECN#, an AQM algorithm designed for the Tofino that targets data center use. Based on the idea of CoDel, ECN# is placed in the egress, and the authors choose their calculation rules with the capabilities of the Tofino in mind. They additionally use several techniques to maximize the number of expressible functionality on Tofino and show a glimpse of the large design space available on ASICs, although it remains largely unexplored.

Possible implementations range from solutions that run entirely in the data plane to those utilizing the CPU managing the ASIC (control plane). While pure data plane implementations run at line-rate, computations are limited and only triggered upon packet arrival. In contrast, the CPU-powered control plane can perform arbitrary computations but cannot provide per-packet calculations due to additional delay and slower processing compared to the data plane. Consequently, there is significant potential in exploring this large design space. While directly designing new algorithms, such as ECN#, is certainly possible, we believe that there is a special value in first investigating how established AQM algorithms can be approximated given the specific constraints of current ASICs.

Choosing a Study Subject. Based on our findings in Table I, RED and CoDel are best suited to match the given restrictions. CoDel can be implemented on Tofino while RED only requires a workaround for the information backflow problem, which also exists for PIE. In contrast, an implementation of PIE is challenging even on a full-fledged Tofino. However, PIE offers some intriguing options for trade-offs that could still enable an approximated implementation. First, the absence of expensive per-packet processing makes it possible to explore the option of moving calculations entirely to the control plane. Second, some complexity of PIE is related to the additional scaling of the probability update and can easily be removed; yet, the impact on performance is unknown and warrants an

	$Delay_{ref}$	T_{Update}	α	β
Default	15 ms	15 ms	0.125	1.25
Adjusted	125 μs	117 μs	0.1171875	157
Scaled	-	-	503	676205
Shifts	-	-	<<9	<<19

TABLE II
PIE PARAMETERIZATION USED IN THIS WORK.

investigation. Third, there are several ways to approximate the probability update calculations on the data plane.

Consequently, PIE is an ideal candidate to explore the large available design space on ASICs; we choose to implement it on the Intel Tofino as we will detail in the following.

VI. IMPLEMENTING PIE ON THE P4-ENABLED TOFINO

The unique combination of capabilities and constraints offers several options for the implementation of PIE. Thus, choosing between the available options can quickly become a challenge. To demonstrate the impact of different constraints and possibilities on AQM design, we have implemented approximated versions of PIE for Tofino using $P4_{16}$ and Intel SDE 9.2.0 in three ways: (1) relying on the control plane (PIE_{CP}), (2) entirely in the data plane using registers (PIE_{DP}), and (3) replacing some of the computations of (2) by table lookups (PIE_{TABLE}). Before describing the designs, constraints, and corresponding trade-offs of our approaches in more detail, we first give an overview of the settings, assumptions used, and challenges faced throughout this work.

A. Overview

Scenario. PIE_{DP} and PIE_{TABLE} use approximations and require tuning to the envisioned RTT spectrum. For our investigation, we choose low RTT scenarios as these are more challenging for AQM design than higher RTTs. Additionally, the buffer sizes of our test switches would be too small to follow common buffer sizing rules when investigating higher RTTs [26].

Assumptions. Our implementations base on two assumptions: Instead of the available nanosecond resolution, we *i)* only use microsecond resolution for the delay measurements as nanosecond fluctuations are hard to grasp given t_{Update} is in the microsecond range (see below); *ii)*, we introduce an artificial queue limit of 500 μs to have a controlled buffer setup.

Setting PIE Parameters. In our local setting, we observe RTTs of 350 μs. Using the guidelines of RFC 8033 [6] (see Sec. II), we choose $delay_{ref}$ as 125 μs and set t_{Update} to 117 μs to allow for computation using the halving rule. Also adjusting the control parameters, we obtain the parameterization shown in Table II. Note that α and β are by default in Hz and need scaling to the Tofino's time-unit in μs; we use the α and β values given in the row labeled with 'scaled'.

Implementation Challenges. Combining the detailed view on PIE with our analysis in Sec. IV, several aspects make a straightforward, naïve implementation of PIE difficult: first, it is not clear how to compute the initial $drop_{prob}$ update as it requires multiplications. Second, the scaling of the $drop_{prob}$ update, as well as the decay of the $drop_{prob}$ itself, require multiple read and write accesses to $drop_{prob}$. When storing $drop_{prob}$ in a register, these multiple accesses, and with them, the scaling and decay are not possible [17].

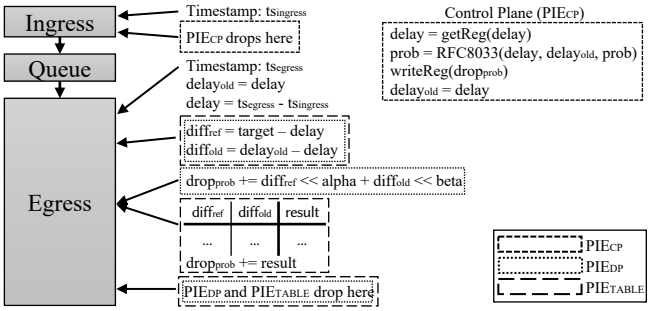

Fig. 3. Three PIE variants: using control and data plane (PIE_{CP}), using stateful memory in the data plane (PIE_{DP}), using tables in the data plane (PIE_{TABLE}).

B. PIE_{CP} — PIE in Control and Data Plane

Even though designed to be simple, PIE comprises several calculation steps that we found hard to map onto our hardware. The switch-local control plane, however, can perform arbitrary computations, although at the cost of significantly slower performance. Thus, it can only be reasonably incorporated for operations at rates much lower than the packet forwarding rate. As this is the case for PIE, which only updates $drop_{prob}$ 2-3 times per RTT, offloading the expensive $drop_{prob}$ computations to the control plane while keeping responsibilities on the ASIC simple is a natural first approach that we follow.

Implementation. In our implementation of PIE_{CP} (illustrated in Fig. 3), the data plane reads $drop_{prob}$ at the ingress and drops a packet accordingly. For packets that are forwarded, it measures the queuing delay and stores it in a per-queue register when the packet enters the egress. The switch-local control plane periodically samples this queuing delay by reading the corresponding registers, updates $drop_{prob}$, and writes the new probability back to the ASIC. The $drop_{prob}$ updates include the basic functionality (Fig. 1 (a)), the scaling ((b)), and the exponential decay ((c)) and are thus according to the RFC. Moreover, our approach uses little resources on the ASIC, leaving plenty of space for other features.

Limitations. Involving the control plane causes $drop_{prob}$ update delays of around 125 μs. For us, this is still fast enough, updating $drop_{prob}$ 2-3 times RTT (as suggested), but this update time is slightly larger than our targeted update time of 117 μs. Regardless, the control plane clearly limits PIE's applicability to higher update rates. Thus, our remaining two approaches try to implement PIE entirely in the data plane.

C. PIE_{DP} — PIE in the Data Plane

The main challenge of implementing PIE in the data plane are the tight computational capabilities: multiplications are not natively supported [15], [16], and stage-local memory can only be accessed within a single MAU [17]. Our implementation, PIE_{DP}, thus deviates from the real PIE (PIE_{RFC}) by simplifying some operations and completely leaving out others.

***$drop_{prob}$* Computation.** The first challenge is that it is not clear how to compute the $drop_{prob}$ update as it requires multiplication (Fig. 1 (a)). PIE_{DP} approximates the computations by expressing the multiplications through bit shifts. For PIE's default

parameterization, there are fitting shifts[1]. However, our scaled α and β parameters are no multiples of two; we resort to the closest power of two, resulting in $<< 9$ for α and $<< 19$ for β. PIE_{RFC} additionally requires a scaling of the $drop_{prob}$ update and an exponential decay of $drop_{prob}$ (Fig. 1 ⓑ/ⓒ), which, together, exceed the number of possible operations within a single MAU and thus require additional accesses to the $drop_{prob}$ register. As these violate the single access rule, PIE_{DP} omits the additional scaling and the exponential decay. Apart from these differences regarding the $drop_{prob}$ calculations, PIE_{DP} deviates from PIE_{RFC} in two other aspects.

Egress Drop. PIE_{RFC} and PIE_{CP} drop in the ingress to reduce the load on the queue. The strict pipeline model of the Tofino, however, disallows feeding information back from the egress to the ingress without actually moving a packet from egress to ingress, e.g., via recirculation. Obtaining the information in the ingress is thus not feasible. As a simple solution, PIE_{DP} performs the dropping directly in the egress.

Sampling. PIE_{RFC} and PIE_{CP} use periodic sampling to further reduce the computational load on the switch. The data plane, however, works on a per-packet basis, and computations are always subject to packet arrivals. To trigger the $drop_{prob}$ updates irrespective of the actual traffic, we take advantage of Tofino's packet generator and create a stream of fixed-period sampling packets that arrive at the desired update intervals.

Limitations. In contrast to PIE_{CP}, PIE_{DP} only provides approximate results for the $drop_{prob}$ computations. The missing $drop_{prob}$ update scaling and exponential decay should make PIE_{DP} more aggressive when congestion is low and slower to react when congestion vanishes. Additionally, our implementation currently uses several sequential computations and occupies 91 % of the overall available pipeline stages for the PIE-specific implementation. While most of these stages are lightly loaded, i.e., they leave plenty of space for other functionality to be executed in parallel, this observation motivates our third approach, PIE_{TABLE}, which intends to reduce the number of occupied stages by using the available tablespace.

D. PIE_{TABLE} — PIE using Tables

Related work [15] has demonstrated that utilizing tablespace to replace some computations in the data plane can help to distribute resource use to different components and improve performance. Following this idea, PIE_{TABLE} replaces PIE_{DP}'s bit shift $drop_{prob}$ approximations with table lookups of pre-computed probability updates (allowing to use the precise multiplicands instead of bit shifts). PIE_{TABLE}'s performance consequently largely depends on the precision of the table entries and thus on the number of stored key-value pairs.

Value Range. Following the $drop_{prob}$ update computation rule in Fig. 1 ⓐ, the lookup table has to cover each possible combination of delay $-$ $delay_{ref}$ ($diff_{ref}$) and delay $-$ $\text{delay}_{\text{old}}$ ($diff_{old}$). Given that we use microsecond timestamps and a queue limit of 500 μs, our setting comprises $501 \times 1001 = 501\,501$ different combinations. While there is enough tablespace available to us, it might not always be feasible to store all combinations, e.g., when using a higher timestamp resolution or queue limit, or when space is needed for the non-AQM operations. Thus, to conserve space, we approximate $drop_{prob}$ update values, e.g., using range matches. Finding the *right* layout for the table, i.e., which key-value pairs to store, then quickly becomes a challenge. In our setting, we choose a maximum table size of 100 000 entries.

Lookup Design. Fundamentally, our implementation performs combined range lookups of $diff_{ref}$ and $diff_{old}$ and returns the corresponding $drop_{prob}$ update value, as illustrated in Fig. 3. As combining two range matches in one lookup is expensive, we further divide the table lookup into two steps. We first perform separate range matches for $diff_{ref}$ and $diff_{old}$ to determine range IDs. We then use these IDs to look up the final $drop_{prob}$ update value with exact matches.

Table Layout. $Diff_{ref}$ can take values from -125 μs to 375 μs, i.e., 501 distinct values, while the range for $diff_{old}$ is from -500 μs to 500 μs, i.e., 1001 distinct values. We map these ranges to our tables as follows. For $diff_{ref}$, we use a resolution of 1 μs between -10 μs and 10 μs (exact match), while we are more imprecise with a granularity of 2 μs for the rest of the spectrum (range match), overall, resulting in 248 values. For $diff_{old}$, we use range matches of 3 μs over the whole spectrum (335 values). In both settings, we represent the value of 0 μs as a distinct exact range. Our final table has roughly 83 000 values for looking up the $drop_{prob}$ update value.

Limitations. There are many possible layouts for the tables, and in- or decreasing the resolution in some ranges impacts the accuracy of the overall computation. When including lookup tables, it is thus essential to carefully choose the key-value pairs; an *optimal* layout is out of the scope of this work. Regardless, it is not straightforward to decide how much of the tablespace can or should be used for the lookups. Additionally, PIE_{TABLE} inherits the main flaws of PIE_{DP}, i.e., it does not include the $drop_{prob}$ update scaling, the exponential decay, and it also drops in the egress. Still, PIE_{TABLE} reduces the number of stages used for the PIE features to 66%.

E. Resource Footprints

Due to the lack of a freely available $P4_{16}$ baseline program, Table III directly compares the ASIC resource use of our three PIE variants. PIE_{CP} uses the fewest resources as most of the computation logic is implemented in the control plane. In contrast, implementing the logic using arithmetic operations in the data plane requires more stages and match crossbars for sequential and conditional processing which is also reflected by a higher share of VLIW actions used for PIE_{DP}. PIE_{TABLE} reduces some of this complexity (fewer VLIW actions and stages) at the cost of higher SRAM and TCAM use.

Resource	PIE_{CP}	PIE_{DP}	PIE_{TABLE}
Stages	33.3 %	91.6 %	66.6 %
SRAM	2.4 %	2.5 %	7.4 %
TCAM	0.0 %	0.3 %	1.3 %
Hash Bits	2.7 %	2.9 %	3.6 %
VLIW Actions	1.8 %	6.0 %	3.6 %
(Exact + Ternary) Match Crossbar	1.6 %	3.4 %	3.6 %
(Stateful + Meter) ALUs	5.2 %	5.2 %	6.3 %

TABLE III
OVERALL RESOURCE FOOTPRINT OF OUR PIE VARIANTS.

[1] α = 1/8 $\hat{=}$ >> 3 and β = 1 + 1/4 $\hat{=}$ >> 2

Fig. 4. The testbed consists of two Tofino switches and two hosts, interconnected at different link speeds.

Takeaway. *PIE_{CP} has a lightweight data plane implementation and performs accurate $drop_{prob}$ computations, although at the cost of CPU usage and limited applicability to low RTTs. PIE_{DP} occupies many stages and approximates the $drop_{prob}$ using bit shifts. It further drops in the egress and neither provides $drop_{prob}$ update scaling nor exponential decay. PIE_{TABLE} reduces the required number of stages by replacing the bit shifts with lookup tables but inherits PIE_{DP}'s shortcomings.*

So far, we have only compared our three PIE flavors qualitatively. To investigate the design impact on the performance, we next compare the behavior of our implementations when facing unresponsive UDP and responsive TCP traffic.

VII. Performance Impact

Our implementations are inherently characterized by design trade-offs. PIE_{DP} and PIE_{TABLE}, e.g., only approximate the $drop_{prob}$ update. To investigate the effects of the trade-offs on the performance, we perform tests with unresponsive UDP and responsive TCP traffic in the testbed shown in Fig. 4. We use two Tofino switches, which are interconnected by a 100 GBit/s link, and two end-host machines. The sender is connected via a 25 GBit/s link, while the receiver is connected with a 10 GBit/s link. Consequently, the link to the receiver is the bottleneck in our system, and a queue will form in the corresponding buffer on Tofino B if there is congestion.

First, using UDP traffic, we evaluate how our implementations increase $drop_{prob}$ under load and how they decrease it when congestion disappears. In a second setting, we multiplex one to 250 TCP connections over the bottleneck and observe if our implementations can actually control the queuing delay. For each of our experimental settings, we perform 30 measurement iterations and show averages and 99% confidence intervals.

A. Unresponsive Traffic

In this experiment, we compare how the different implementations increase and decrease $drop_{prob}$. For this, we send 100 GBit/s of unresponsive UDP traffic from Tofino A to the receiver and investigate $drop_{prob}$ on Tofino B while instructing the AQM algorithms to not drop any packet. This way, we can observe how $drop_{prob}$ increases subject to persistent congestion. When $drop_{prob}$ would drop all packets, we reduce the traffic to 5 GBit/s, i.e., below the bottleneck bandwidth. The queue will drain and $drop_{prob}$ should now decrease.

For each of our PIE flavors, Fig. 5 shows the time required to first increase $drop_{prob}$ to 100% (Rise) and then decrease it back down to 0% (Fall). We additionally compare the results to two ns-3.31 simulations (modified to also not drop) of PIE: $PIE_{ns3\text{-}rfc}$ scales $drop_{prob}$ according to the RFC (see Fig. 1 (b)); $PIE_{ns3\text{-}default}$ uses the scaling as implemented in ns-3, which is more aggressive than $PIE_{ns3\text{-}rfc}$ and increases PIE's $drop_{prob}$ update if the $drop_{prob}$ is greater than 0.1.

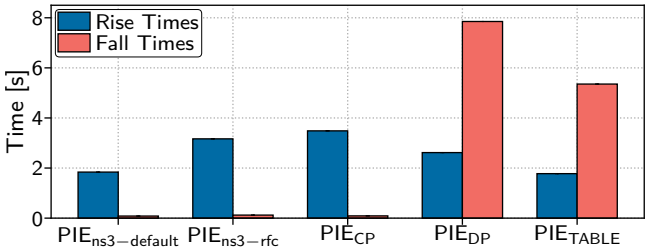

Fig. 5. Durations needed by our three PIE flavors and two ns-3 simulations to reach 100% drop probability (Rise) under full load and to go back to 0% once the load is gone (Fall). Note: the confidence intervals are hardly discernible.

Results. The rise times of all our PIE variants compare reasonably well to the simulation results of roughly 2 s to 3 s. PIE_{CP} takes the longest to reach 100% drop probability as it scales $drop_{prob}$ (see Fig. 1 (b)) to slow down the initial increase. Remarkably, it is slower than $PIE_{ns3\text{-}rfc}$ and $PIE_{ns3\text{-}default}$. We suspect that this is due to its update rate, which is slightly lower than it should be. In contrast, PIE_{DP} and PIE_{TABLE} do not scale $drop_{prob}$ and thus have faster rise times. Further, PIE_{TABLE} has a faster increase than PIE_{DP}, which we attribute to the approximated bit-shifts in PIE_{DP} that are less precise than the approximated lookup tables (using precise multiplicands).

Similarly, PIE_{TABLE} decreases the drop probability slightly faster than PIE_{DP} when we lift the congestion. In comparison, the other variants are much quicker thanks to their exponential decay (see Fig. 1 (c)). Yet, PIE_{CP} again slightly deviates from the behavior of $PIE_{ns3\text{-}rfc}$ as it decreases $drop_{prob}$ faster.

Already from this simple experiment, we infer that design choices, such as removing the $drop_{prob}$ scaling, significantly affect the performance. While persistent congestion is well-suited to compare the basic mechanisms, it fails to illustrate if the implementations can effectively manage the congestion subject to responsive traffic, which we investigate next.

B. Responsive TCP Traffic

In our second experiment, we instruct the AQM algorithms to actually drop packets and investigate whether our implementations can control the queuing delay. Using our testbed, we send 10 s of one to 250 concurrent TCP CUBIC streams from the TCP Sender to the receiving host. Fig. 6 reports on the combined TCP goodput as well as the queuing delay.

Results. Looking at the goodput, PIE_{CP} outperforms the other two Tofino variants and is close to line-rate. This, however, comes at the cost of high queuing delays as it fails to bring the delay down to $delay_{ref}$. In contrast, PIE_{TABLE} allows almost no goodput, and PIE_{DP} only reaches 50% of line-rate with an increasing number of flows. They both seem to be too aggressive as the delay is way below $delay_{ref}$ most of the time. We again attribute these observations to the characteristics of our implementations as the more reasonable results of $PIE_{ns3\text{-}rfc}$ and $PIE_{ns3\text{-}default}$ indicate that our PIE parameterization is generally capable of better performance in our setting.

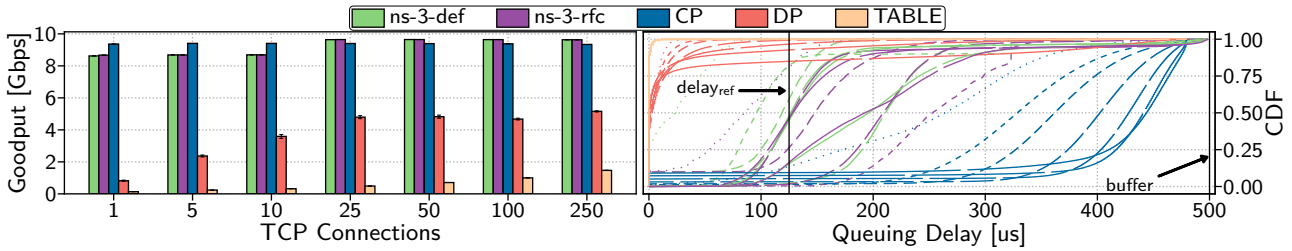

Fig. 6. Average goodput and CDF for the queuing delay for different numbers of concurrent flows (more continuous lines (right plot) represents higher concurrency) passing through our three PIE flavors. Note that most confidence intervals are hardly discernible.

Takeaway. *While PIE_{CP} cannot control the queuing delay, PIE_{DP} and PIE_{TABLE} are too aggressive and overcontrol the queue. We suspect that this is partly due to the missing scaling of the $drop_{prob}$ update. Additionally, the parameterization of α and β can also be a factor as our simulation results leave room for improvement. Although we follow the RFC guidelines, we note that it is not unambiguously defined how to adjust the parameters to a change in t_{Update}. Consequently, we draw two conclusions:* i) *as the inherent characteristics of our implementations are reasoned by the specific constraints of pipeline-based switching ASICs, AQM standardization currently lacks in incorporating these hardware platforms into their designs.* ii) *AQM standardization currently lacks fail-safe parameterization guidelines.* In the following, we further elaborate on the implications of our findings on AQM design.

VIII. Implications for AQM/ASIC Design

Based on our analysis in Sec. IV and our experiments in Sec. VII, we first identify several general implications for the design of AQM algorithms for switching ASICs. We then discuss how ASIC-side support for AQM can be improved.

Information and Processing Model. First, the AQM should match the available information and processing model of the ASIC. Currently, *queue state information* cannot be gathered at the ingress before packet enqueue, and feeding this information back to the ingress is infeasible. Consequently, AQM algorithms should preferably work in the egress.

Complexity. Second, calculation rules for determining the feedback of the AQM should be expressible on the ASIC. This is a two-fold challenge as i) calculations are generally constrained, and ii) as there are strict memory access rules due to the stage-locality of memory. Thus, the operations that make up the control-law should be simple, i.e., they should rely on functionality that is typically used in packet processing, e.g., no multiplications. Further, concepts challenging the stage-locality, such as the scaling of $drop_{prob}$ in PIE, should be avoided.

Applicability. Third, the scaling of system parameters to other network conditions should be straightforward and feasible. For example, PIE's default parameterization can easily be expressed with bit shifts, while the RFC tuning guidelines do not generate fitting bit shifts in our setting. Thus, the AQM should be either tuning-free or have concise and precise ways to adopt parameters to new settings.

Hardware-first Design. Fourth, AQM algorithms should fully capitalize on the line-rate processing capabilities of ASICs. In contrast to software switches, where sampling, like employed in PIE, makes sense for load reduction, ASICs are capable of constant line-rate processing. As such, sampling techniques actually increase the complexity of the approaches, either because they require external triggers or a data plane sampling logic or because one needs to involve the CPU, which adds another (unstable) component to the solution.

Implications for ASIC Design. There are also some aspects of ASIC design that can improve the implementability of AQM. Providing *queue state information* at the ingress before packet enqueue generally enables the native implementation of AQM algorithms such as RED and PIE that are located in the ingress. Additionally, more flexible memory access structures, allowing one early read access and then one later write access, would allow for the calculation of more complex control rules. Ideally, there should be standard ways of providing such functionality so that AQM is easier to deploy across a wide range of ASICs and not limited to platform-specific variants. Finally, making the queue management accessible and programmable via P4 brings the potential of not only indirectly managing the queue, as current approaches do, but of directly managing it.

IX. Conclusion

Current RFC-standardized AQM algorithms are often hard, if not impossible, to implement on programmable, pipeline-based ASICs, challenging the future management of congestion at high-speed links. In this work, we demonstrate these challenges by first analyzing how the design of popular AQM algorithms contradicts the operation model and capabilities of a state-of-the-art networking ASIC. We then implement three flavors of the PIE AQM on an Intel Tofino switch to demonstrate the possible implementation paths, challenges, and trade-offs on this architecture when using P4. Our results further indicate that design choices based on these constraints can have a significant performance impact as none of our three implementations successfully controls the queuing delay. Trade-offs regarding the computation accuracy and the supported AQM features can lead to a severe overcontrolling of the queue, while a too low update rate can cause the opposite. We thus believe that there is a need for AQM research and specifications that are tailored to programable hardware. Lastly, congestion management should become a core feature of ASICs and P4, demanding actual support within the chips and the language, to, e.g., program the traffic manager.

Acknowledgment

We thank Vladimir Gurevich for his insightful feedback and the fruitful discussions. We also thank the anonymous reviewers. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2023 Internet of Production – 390621612.

References

[1] S. Bauer, D. Clark, and W. Lehr, "The Evolution of Internet Congestion," in *Research Conference on Communication, Information and Internet Policy (TPRC)*, 2009.

[2] A. Dhamdhere, D. D. Clark, A. Gamero-Garrido, M. Luckie, R. K. P. Mok, G. Akiwate, K. Gogia, V. Bajpai, A. C. Snoeren, and K. Claffy, "Inferring Persistent Interdomain Congestion," in *ACM SIGCOMM*, 2018, pp. 1—15.

[3] H. Wu, Z. Feng, C. Guo, and Y. Zhang, "ICTCP: Incast Congestion Control for TCP in Data-Center Networks," *IEEE/ACM Transactions on Networking*, vol. 21, no. 2, pp. 345–358, 2012.

[4] J. Gettys and K. Nichols, "Bufferbloat: Dark Buffers in the Internet," *ACM Queue*, vol. 9, no. 11, pp. 40–54, Nov. 2011.

[5] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar, "Controlled Delay Active Queue Management," Internet Requests for Comments, RFC 8289, 2018.

[6] R. Pan, P. Natarajan, F. Baker, and G. White, "Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem," Internet Requests for Comments, RFC 8033, 2017.

[7] T. Høiland-Jørgensen, D. Täht, and J. Morton, "Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways," in *IEEE International Symposium on Local and Metropolitan Area Networks*, 2018, pp. 37–42.

[8] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo, "An Experimental Study of Home Gateway Characteristics," in *ACM Internet Measurement Conference*, 2010, pp. 260–266.

[9] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[10] R. Adams, "Active Queue Management: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 3, pp. 1425–1476, 2012.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[12] R. Kundel, J. Blendin, T. Viernickel, B. Koldehofe, and R. Steinmetz, "P4-CoDel: Active Queue Management in Programmable Data Planes," in *IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2018, pp. 1–4.

[13] S. Laki, P. Vörös, and F. Fejes, "Towards an AQM Evaluation Testbed with P4 and DPDK," in *ACM SIGCOMM Posters and Demos*, 2019, pp. 148—150.

[14] M. Menth, H. Mostafaei, D. Merling, and M. Häberle, "Implementation and Evaluation of Activity-Based Congestion Management Using P4 (P4-ABC)," *Future Internet*, vol. 11, no. 7, pp. 1–12, 2019.

[15] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the Power of Flexible Packet Processing for Network Resource Allocation," in *USENIX Symposium on Networked Systems Design and Implementation*, 2017, pp. 67–82.

[16] S. Wang, J. Bi, C. Sun, and Y. Zhou, "Prophet: Real-time Queue Length Inference in Programmable Switches," in *ACM Symposium on SDN Research*, 2019, pp. 158–159.

[17] J. Zhang, W. Bai, and K. Chen, "Enabling ECN for Datacenter Networks with RTT Variations," in *ACM Conference on emerging Networking EXperiments and Technologies*, 2019, pp. 233–245.

[18] Kunze, Ike and Gunz, Moritz, "PIE for Tofino on GitHub." [Online]. Available: https://github.com/COMSYS/pie-for-tofino

[19] B. Briscoe, K. De Schepper, M. Bagnulo Braun, and G. White, "Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture," Internet-Draft https://datatracker.ietf.org/doc/html/draftietf-tsvwg-l4s-arch-08, Work in Progress, 2020.

[20] J. Morton, P. Heist, and R. Grimes, "The Some Congestion Experienced ECN Codepoint," Internet-Draft https://datatracker.ietf.org/doc/html/draft-morton-tsvwg-sce-02, Work in Progress, 2020.

[21] K. Nichols and V. Jacobson, "Controlling Queue Delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, 2012.

[22] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg, "PIE: A Lightweight Control Scheme to Address the Bufferbloat Problem," in *IEEE International Conference on High Performance Switching and Routing*, 2013, pp. 148–155.

[23] Intel, "Intel® Tofino™," 2021. [Online]. Available: https://https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html

[24] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[25] Â. C. Lapolli, J. Adilson Marques, and L. P. Gaspary, "Offloading Real-time DDoS Attack Detection to Programmable Data Planes," in *IFIP/IEEE Symposium on Integrated Network and Service Management*, 2019, pp. 19–27.

[26] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing Router Buffers," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 281–292, 2004.