`CoDel-ACT`: Realizing CoDel AQM for Programmable Switch ASIC

Vedant Bothra*, Aditya Peer*, Vijay Kumar Singh, Mukulika Maity, Rinku Shah
Department of Computer Science and Engineering
Indraprastha Institute of Information Technology Delhi (IIIT-Delhi)
{vedant20260, aditya20355, vijaysi, mukulika, rinku}@iiitd.ac.in

***Abstract*—Bufferbloat is a major issue in computer networks, caused by network devices such as switches and routers with large buffers, resulting in congested queues and high network delays. Active Queue Management (AQM) algorithms like RED, CoDel, and PIE have been developed to address Bufferbloat. Among these, CoDel is the most effective as it does not require parameter tuning. However, software-based AQM solutions cannot scale for high-speed networks, while traditional hardware switches are fixed-function. Programmable switch ASICs offer more flexibility but impose programming constraints to achieve line rates. Due to these limitations, previous research has been unsuccessful in implementing an RFC-compliant CoDel AQM on programmable switch ASICs. To solve this challenge, we have introduced `CoDel-ACT`, a redesign of CoDel that can handle the global state across switch pipeline stages, improves the accuracy of the math function, and fully implements CoDel within the data plane of an Intel Tofino switch. Our evaluations demonstrate that `CoDel-ACT` solves Bufferbloat, has negligible overheads, and reduces the average packet queue delay by 52% compared to the existing solution.**

***Index Terms*—Bufferbloat, AQM, CoDel, P4 switch ASIC**

I. Introduction

Network routers have been equipped with buffers that can absorb short-term data bursts and enable faster data transmission. However, these buffers introduce queuing delays and delay variations in the network. When the buffers become too full, packets are lost; when they are too empty, the network throughput can be degraded.

The Bufferbloat Problem. The significant distance between the sender and receiver in internet traffic (10ms to 100ms RTT [1]) can create slow sender reactions to congestion. When network links are overutilized, acknowledgments can be delayed beyond the round-trip time, creating a "persistently full" queue. Such a queue can contribute to additional network delays and may not be able to absorb bursts if it never drains completely, causing "Bufferbloat" [2].

Active Queue Management (AQM) algorithms such as RED [3], CoDel [1], and PIE [4] aim to differentiate between short bursts and congestion to avoid persistently full queues by dropping packets before the buffer overloads. The effectiveness of RED and PIE depends on the algorithm parameter values, which are difficult to tune accurately for each network scenario, while CoDel is designed to be parameterless.

CoDel (Controlled Delay). CoDel [1] aims to keep the queuing delay (time spent by a packet within the switch buffers) below the maximum acceptable persistent queue delay, defined by *TARGET*. If the delay exceeds this value for a prolonged time (defined by *INTERVAL*), CoDel starts dropping packets (see §III). It schedules further packet drops in shorter succession using $\textit{dropNext} = \textit{now} + \frac{\textit{INTERVAL}}{\sqrt{count}}$ until the observed queue delay falls below the *TARGET*. Here, $dropNext$ is the time to drop the next packet, and $count$ refers to the number of dropped packets during the current congestion cycle. During prolonged congestion, CoDel uses the drop rate from the previous cycles as a starting point for the current cycle, helping to quickly resolve network congestion by an aggressive increase in packet drop rate, i.e., $count_{curr} = count_{curr-1} - count_{curr-2}$.

Existing AQM implementations. Congestion can occur at different points in a network, including the low-bandwidth last-mile links, fat backbone links, or data centers.

① Last-mile gateways can use *software-based* AQM solutions [1], [5]–[9] implemented for Linux kernel or software switches such as bmv2 [10] and DPDK [7]. However, software-based solutions cannot handle high-speed backbone networks with traffic speeds of several hundred Gbps. ② High-speed backbone networks use *specialized hardware* with built-in AQM [11]–[14] algorithms that are specialized based on workload requirements. Several switch platforms such as Arista (Trident and Tomahawk, Trident II and Helix) [12], and Cisco [11] implement Weighted Random Early Detection (WRED) algorithm in their firmware. Cable modem [13] implements modified CoDel and PIE AQM to satisfy cable traffic requirements. CAKE AQM [14] was designed for home gateways as part of the OpenWrt router firmware. However, specialized AQM is necessary based on workload characteristics, driving the need for programmable network hardware solutions. ③ Modern data centers and ISPs use *programmable network hardware* to develop and deploy custom programs that run at high speeds. Sivaraman *et al.* [15] added an FPGA to the P4-based programmable Tofino switch [16] to fully implement CoDel and RED AQM algorithm variants. Using an additional off-path FPGA introduces additional latencies, resulting in an increase in flow RTT. `CoDel-ACT` implements CoDel entirely in the Tofino switch data plane without depending on additional hardware support. The work closest to

* Student authors with equal contribution.

ISBN 978-3-903176-63-8 © 2024 IFIP

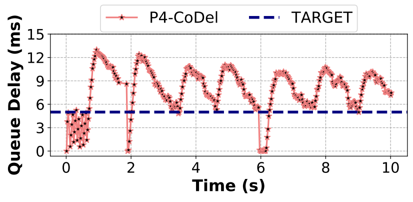

Figure 1. Measured queue delay for existing Intel Tofino switch ASIC-based CoDel solution for *ten* parallel TCP flows is $8.3ms$

CoDel-ACT, Kundel *et al.* [17], implemented partial CoDel AQM implementation for the Tofino switch, leading to sub-optimal performance.

Can we implement custom AQM on smartNICs? SmartNIC-based custom AQM can handle bufferbloat for the ingress traffic at the host and can be implemented at the ingress gateway host in data centers. Du *et al.* [18] used NetFPGA NIC to implement custom AQM, R-AQM, that improved the tenant's TCP performance at the virtualized data center servers. Ralph *et al.* [17]'s CoDel experienced high flow RTT due to Netronome smartNIC's [19] many-core processor architecture. SmartNIC-based off-path processing results in additional latencies — not the right candidate for resolving network-wide bufferbloat.

Tofino switch is used for data center switching. Is AQM required in data centers? Data center switches are equipped with shallow buffers to control queueing delays and ensure under millisecond flow RTT [20]. However, high-speed traffic with diverse mix of short, long, and bursty flows leads to problems such as the *TCP incast* [21], [22] and *HOL blocking* due to elephant flows [23], raising the need for congestion control and AQM implementation [24], [25] in data centers.

Does existing programmable switch-based CoDel design effectively solve Bufferbloat? We conducted experiments using the open source CoDel code, *P4-CoDel* [26], implemented on the Intel Tofino switch (setup details in §VI). We set the *TARGET* to 5 ms in accordance with CoDel's RFC. Fig. 1 shows the per packet queueing delay for *ten* parallel TCP flows. We noticed that the packet delays for the P4-CoDel design were mostly above the TARGET, and the average queuing delay was $8.3ms$. This indicates that P4-CoDel is not fully capable of solving the Bufferbloat problem. This is because the design does not maintain the packet drop count from previous cycles and starts with $count = 1$, which results in a longer reaction time.

In this paper, we design and implement CoDel-ACT, where ACT stands for *A*dapted *C*ount-and-time for *T*ofino switch ASIC. CoDel-ACT adapts the packet dropping rate based on the *packet drop count* history, as outlined in the RFC. It operates at line rates and runs entirely in the data plane of the Tofino switch ASIC.

Challenges. There were several challenges in realizing CoDel for programmable switch ASIC. ① Programmable switch ASIC pipelines force certain programming constraints to ensure line-rate packet processing (see §II-B). Due to this, accessing the packet drop count of the previous cycles stored in Tofino registers between CoDel cycles is non-trivial. ② CoDel algorithm uses the function, $\frac{INTERVAL}{\sqrt{count}}$ to compute $dropNext$, that determines how quickly CoDel reacts to congestion. However, the existing approximation of this function [17] for Intel Tofino was overestimated and varied by 42% to 58%, resulting in a longer reaction time.

We redesigned the CoDel algorithm to split the *count* state (and other related states) across multiple stateful registers to make it amenable for programmable switch ASIC implementation (see §V-A) and recirculated the required state within the switch egress pipeline. We designed an improved approximation of the mathematical function to compute $dropNext$. Our approximation has an average error rate of 4% and a maximum error rate of 15%. We implemented CoDel-ACT using 11 pipeline stages of the Intel Tofino switch and was programmed using P4-16 and Intel SDE 9.9.0. The code for CoDel-ACT is open source [1].

Contributions. The main contributions of this paper are as follows. ① We present CoDel-ACT, a redesigned CoDel AQM, a first programmable switch ASIC solution that closely represents the CoDel algorithm as described in RFC [1] and runs entirely in the data plane. ② We improve the accuracy of the CoDel function that computes the packet dropping rate for Intel Tofino switch ASIC. This enhancement enables CoDel-ACT to react more aggressively to congestion. ③ We compare the effectiveness of CoDel-ACT with the existing Tofino-based CoDel solution, P4-CoDel.

Our evaluations show that CoDel-ACT maintains the average queue delay well below *TARGET*, reduces average packet queue delay by 52% and tail latency by 50% for 50 concurrent TCP flows with bottleneck bandwidth of 1Gbps, compared to P4-CoDel.

The rest of the paper is organized as follows. §II and §III describe the background on programmable switch ASICs and CoDel RFC; §IV discusses the challenges in realizing CoDel for Intel Tofino switch; §V discusses CoDel-ACT's design and implementation, and §VI demonstrates CoDel-ACT's efficacy compared to the existing work.

II. Programmable switch ASICs

The ASIC-powered switches offer support for the P4 language, which allows the programmer to define switch forwarding behavior. In this section, we briefly discuss the programmable switch architecture and the programming constraints of the Intel Tofino switch ASIC [16].

A. Programmable switch ASIC architecture

Fig. 2 shows the pipeline design of an ASIC-based programmable switch. The pipeline comprises programmable packet parsers and de-parsers, an egress with several stages of programmable Match Action Units (MAUs), stateful memory, a buffer, a non-programmable Traffic Manager (TM), and a similarly structured ingress pipeline. Stateful memory such as *registers* is physically divided across switch pipeline stages.

[1]https://github.com/pnl-iiitd/codelACT

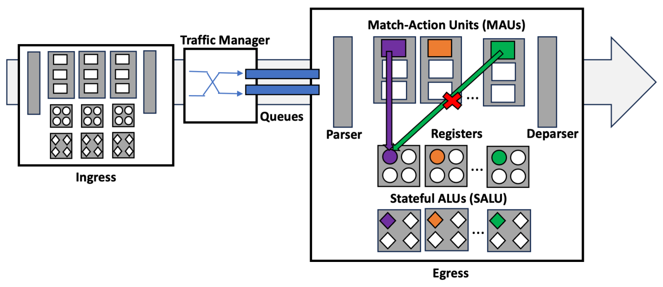

Figure 2. Programmable ASIC-based switch architecture

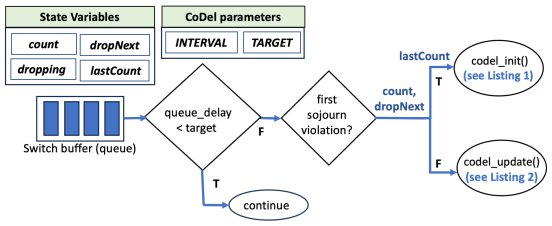

Figure 3. CoDel's flow diagram, based on RFC

Tofino switch ASIC allows the P4 program to operate atomically on the registers using a small number of instructions within a single pipeline stage using stateful ALUs ($SALU$).

B. Constraints for the Intel Tofino switch

To achieve line rates of several Tbps, programmable switch ASICs allow simple operations that can be executed within deterministic time delay.
① **Packet processing constraints.** ⓐ The ALU instructions are limited to signed addition and bitwise logical operations. Multiplication, division, floating-point computations, and pointers are not allowed. ⓑ Control flow mechanisms, such as loops, are prohibited. Additional mechanisms such as hashing primitives are provided as *externs*.
② **Stateful memory constraints.** ⓐ Register action guarantees atomic operations on packet fields. It does not allow computing with inputs such as packet metadata and math-unit output. ⓑ Each switch pipeline stage can execute only one ALU instruction per packet field. The switch pipeline limits the number of sequential processing steps to the number of match+action pipeline stages (12 for our switch). ⓒ Tofino switch memory (i.e., tables and registers) is statically assigned to each MAU. Therefore, a packet can access this memory only once per pipeline pass. ⓓ Stateful elements, such as the registers, can be accessed atomically but are limited to one read-update-write operation. Tofino supports more expressive update functionality using a simple microprogram, *register action extern*.

III. CoDel algorithm described in RFC

In this section, we present the working of the CoDel AQM as described in the RFC [1]. CoDel uses two input parameters, *INTERVAL* and *TARGET*. The *INTERVAL* value is chosen such that the endpoints have enough time to respond to packet drops, i.e., slightly higher than RTT ($100ms$ for Internet). The maximum acceptable persistent queue delay, *TARGET*, is set between $5\% - 10\%$ of the *INTERVAL*. CoDel algorithm includes four "global" state variables: 1) *dropping* (boolean, if true indicates that CoDel may drop packets), 2) *count* (packet drop count of the current congestion cycle), 3) *lastCount* (packet drop count of the previous cycle) and 4) *dropNext* (time when the packet should be dropped next).

Fig. 3 shows the packet flow followed by each outgoing packet of a switch that implements CoDel. The algorithm observes the *sojourn time* (time spent by the packet in the switch buffer) for each outgoing packet. If the *sojourn time* exceeds the *TARGET* (i.e., sojourn violation), it implies that the queue is getting filled, and the system stays in the *dropping* state. Otherwise, the system is in non-dropping state ($dropping = false$). If it is a first sojourn violation, codel_init() is invoked, to initialise the CoDel state variables. Otherwise, codel_update() is invoked.

```
1  define INTERVAL 100 ms
2  # Here, dropping = True
3  # lastCount: packet drop count for cycle, i-2
4  # count: packet drop count for cycle, i-1
5  # dropNext: time when to drop next packet
6  # now: curr time; use packet's egress timestamp
7  delta = count - lastCount
8  lastCount = count
9  if (delta>1) and (now-dropNext<16*INTERVAL):
10     count = delta
11 else:
12     count = 1
13 dropNext = now + INTERVAL/sqrt(count)
```

Listing 1. CoDel RFC: codel_init() function

```
1  # Here, dropping = True.
2  # Sender should have reacted to pkt drop by now
3  if (now >= dropNext):
4      # Drop the packet
5      count = count + 1   # Update "count"
6      dropNext += INTERVAL/sqrt(count)
```

Listing 2. CoDel RFC: codel_update() function

codel_init function: In this function, all the state variables are initialized (see Listing 1). To recover from prolonged congestion ($delta > 1$), CoDel aggressively drops packets. If the queue was detected full recently (line 9), then the drop rate that controlled the queue for that cycle is a good starting point for the current cycle; *count* is set to *delta*. Otherwise, *count* is set to 1. The historic *count* state is stored as *lastCount* for the next congestion cycle (line 8).
codel_update function: In this function, CoDel starts to drop packets. Before dropping a packet, CoDel waits for the sender to react to the previous packet drop (see Listing 2, line 3). If the packet is dropped, *count* is incremented, and the time to drop the next packet *dropNext* is updated (lines 4 to 6).

IV. Challenges in implementing CoDel for Tofino switch ASIC

A. Can we implement CoDel-RFC for Tofino switch ASIC?

Fig. 4 shows the hypothetical design for RFC-based CoDel offloaded to Tofino switch ASIC. It shows the program flow for

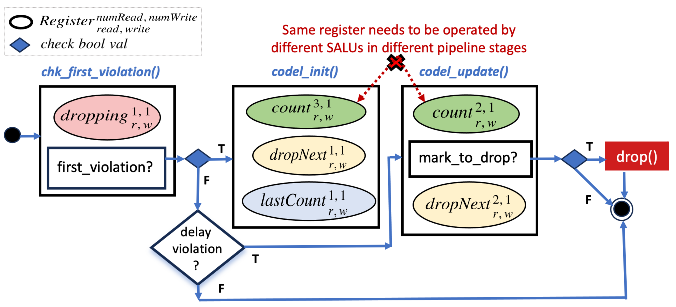

Figure 4. Hypothetical CoDel-RFC design for Tofino switch ASIC

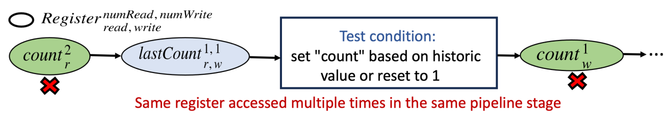
Register numRead, numWrite read, write
count 2 r
lastCount 1, 1 r, w
Test condition: set "count" based on historic value or reset to 1
count 1 w
Same register accessed multiple times in the same pipeline stage

Figure 5. Partial flow for hypothetical *codel_init* function

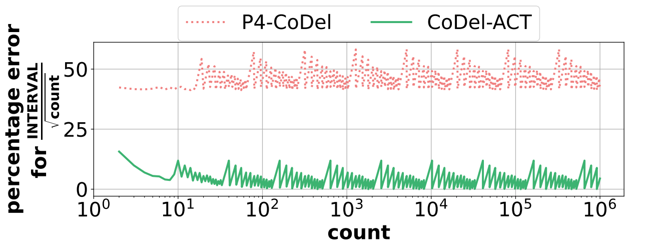

Figure 6. Percentage error rate for measured $\frac{INTERVAL}{\sqrt{count}}$ relative to actual value computed using math.sqrt()

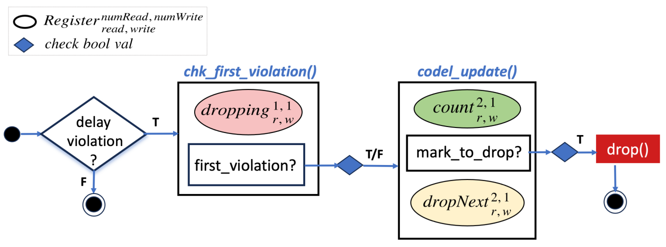

Figure 7. Flow diagram of existing Tofino-based CoDel implementation (P4-CoDel); function *codel_init*() implementation is skipped

three CoDel functions, viz., chk_first_violation(), codel_init(), and codel_update(), and the registers accessed by each function. The registers store the "global" state variables (see §III), *dropping*, *count*, *dropNext*, and *lastCount*.

As per the current design, it is not possible to implement *codel_init()* and *codel_update()* within the same P4 program for the Tofino switch ASIC due to the following reasons:
① ***A register cannot be accessed across pipeline stages*** (§II-B(2c)). Though codel_init() and codel_update() are independent functions, they cannot run in parallel since they share common state. codel_init() first accesses *count* and then *dropNext* (see Listing 1 lines 7–9), whereas codel_update() accesses *dropNext* first, and then *count* (see Listing 2 lines 3–5). These functions are processed using separate SALUs, which resembles the scenario shown in Fig. 9 and Fig. 10. Register *count* is accessed in $stage_i$ for codel_init() but a different stage for codel_update(); the same holds for register, *dropNext*. Even if the access order were the same, the same registers would be accessed by different pipeline stages, which is not permitted.
② ***A packet can access any register only once within the pipeline (§II-B(2d))***. We observe challenges within the codel_init() implementation as well. Fig. 5 shows the codel_init() function flow. First, registers *count* and *lastCount* are read, after which a condition is tested that requires the math function (Listing 1, line 9). This requires the program to exit SALU that accesses *count* and *lastCount*. After the condition test, *count* must be accordingly updated (Listing 1, line 10 & 12). However, since a packet can access the register only once, codel_init() implementation is not feasible.
③ ***High error rate for dropNext computation.*** To update the time to drop the next packet (*dropNext*), $\frac{INTERVAL}{\sqrt{count}}$ is computed using Tofino's in-built math unit. We observe that the computed value is higher by *42% to 58% (see Fig. 6)* compared to the actual values, making CoDel less aggressive.

B. *Existing Tofino-based CoDel implementation*

Fig. 7 shows the P4-CoDel's [17] design to offload CoDel to Intel Tofino switch. They use two SALUs: (1) to implement *chk_first_violation()*, and (2) to implement *codel_update()*. The implementation is feasible since both SALUs use independent registers. Skipping *codel_init()* implementation and the higher $\frac{INTERVAL}{\sqrt{count}}$ value (Tofino's math unit) leads to a less aggressive CoDel implementation, and a *persistently full queue* during prolonged congestion periods (as shown in Fig. 1).

V. Design and Implementation

In this section, we discuss how `CoDel-ACT`'s design helps resolve the Tofino switch offload challenges discussed in §IV.

A. `CoDel-ACT` *Design and Implementation*

CoDel observes the packet queuing delay to detect sojourn violations. Since this information is available at the switch egress, `CoDel-ACT` runs within the switch egress pipeline. To make CoDel compatible with Tofino switch ASICs, `CoDel-ACT` makes use of register copies (we call them *shadow registers*) within the CoDel function and across functions. Our design ensures synchronization across the register copies using packet recirculation, which involves recirculating the packet from the *switch egress* back to the *Traffic manager*.

Fig. 8 shows the flow diagram of the proposed `CoDel-ACT` design that implements complete CoDel functionality as mentioned in the RFC [1] for Tofino switch ASIC. `CoDel-ACT` maintains *four* state variables, *dropping*, *count*, *dropNext*, and *lastCount* as discussed in §III. To handle multiple register accesses within and between pipeline stages, we use three register copies for *count* (viz., *countI*, *countI'*, and *countU*) and *dropNext* (viz., *dropNextI*, *dropNextI'*, and *dropNextU*) and two copies for *dropping* (viz., *dropping* and *prevDropping*). The switch pipeline design comprises two parallel pipelines that process: (1) synchronization traffic, `CoDel-ACT` *packet* and (2) incoming data traffic, i.e., *data* packet.

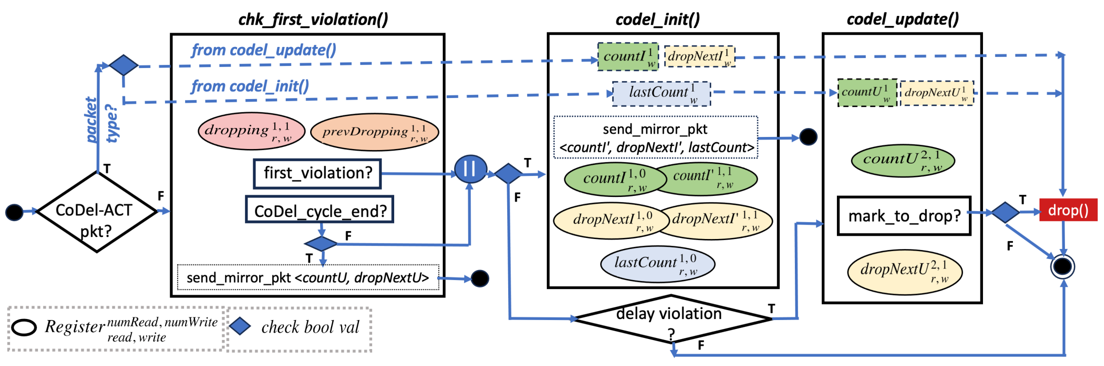

Figure 8. Flow diagram for CoDel-ACT offloaded to Tofino switch

```
if(some_condition):
    R1.action() #stage i
    R2.action() #stage i+1
else:
    R2.action() #stage i
    R1.action() #stage i+1
```

Figure 9. Stage constraint

```
if(variable >= A):
    R1.read()
if(variable >= B):
    R1.read()
```

Figure 10. Access constraint

Each *data* packet is first processed by $chk\_first\_violation()$, which checks if it is the first sojourn violation, accordingly sets the *dropping* state, and synchronizes *codel_update()*'s register state with *codel_init()* if the congestion cycle just ended. We monitor the dropping state during the current (*dropping*) and the previous packet (*prevDropping*) processing to identify the end of the congestion cycle.

Next, we discuss how we solve the primary challenges in making CoDel amenable for Tofino switch offload (see §IV-A).

① **Shared access to registers, *count* and *dropNext*, between *codel_init()* and *codel_update()* functions.** A packet can either invoke *codel_init()* or *codel_update()* based on whether it was the first sojourn violation or not. In case of first sojourn violation, codel_init() is invoked exactly once; codel_update() function operates only after $\frac{INTERVAL}{\sqrt{count}}$ (i.e., upto $100ms$). The codel_init() function is reinvoked during the next congestion cycle, and there is ample time gap between consecutive function invocations.

The key idea of our approach is to use separate registers, viz., $countI$ and $countU$ for codel_init() and codel_update(), respectively. The registers are synchronized at the start of each congestion cycle when codel_init() is invoked and at the end when *dropping* turns $false$, using packet recirculation. Packet mirroring is used in Tofino switch to implement packet recirculation. The recirculated (or mirror) packet carries the register values of the CoDel function and is sent back to the switch egress pipeline via the Traffic manager. The mirror packet from codel_init() is processed by codel_update() and vice versa. This is shown as a dashed flow in Fig. 8.

② **Allowing a packet to access the same register more than once during a single pass through the switch pipeline.** Within *codel_init()*, the registers *countI* and *dropNextI* are first read, and their update operation is interleaved with another instruction (discussed in §IV-A). Therefore, within *codel_init()*, shadow registers, $countI'$ and $dropNextI'$, are updated, and the original registers are updated via $codel\_update()$'s mirror packet at the end of the current congestion cycle.

Similarly, consider the following processing sequence: $count.read() \rightarrow lastCount.read() \rightarrow updatedelta \rightarrow lastCount.write()$. We observe that $lastCount$ read and update operations would require accessing the same register twice, which is not permitted. Therefore, the $lastCount.write()$ is achieved using the mirror packet generated by $codel\_init()$ (shown as $dashed$ flow in Fig. 8).

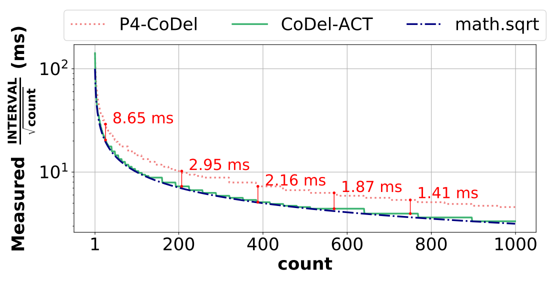

Figure 11. Absolute error for $dropNext$ calculation: P4-CoDel delays packet dropping by up to $30ms$; $count = 2$; CoDel-ACT delays packet dropping by up to $11ms$; $count = 3$

③ **Reducing error rate for** $dropNext$ **computation.** Our measurements (see Fig. 11) show that the existing Tofino's math unit overestimates the $\frac{INTERVAL}{\sqrt{count}}$ by up to $30ms$, indicating that no packets would be dropped for next $30ms$ in the worst case, leading to a persistently full queue. We observed that a relation exists between the error rate and $count$. If we double the value of $count$, the error rate drops significantly. Therefore, to compute $\frac{INTERVAL}{\sqrt{count}}$, we provided the input as *2*count* to the math unit. To achieve this, we modified the CoDel implementation (Listing 2's line 5) to increment $count$ by two, resulting in an average error rate of 4% (see Fig. 6).

Implementation details. We implemented the RFC-compliant CoDel algorithm within the egress pipeline of Aurora 610, Intel Tofino switch, which spanned around 400 lines of P4 (P4-16 version) code. We used *11* pipeline stages to realize packet processing at line rates.

VI. Evaluation

In this section, we will describe our experimental setup, present the evaluation metrics, and formulate the research questions. Finally, we will discuss the evaluation results.

Setup. We set up a testbed with one *AMD Ryzen 9 5950X* workstation and an *Aurora 610 Intel Tofino switch*, as shown in Fig. 15. To generate traffic, we used *iperf3* to create parallel TCP flows from the workstation acting as the TCP client and

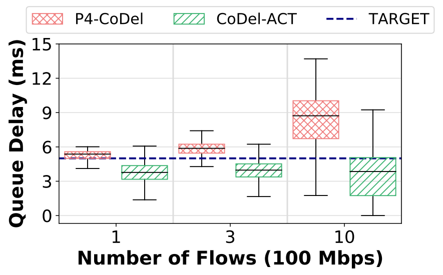

Figure 12. Queue Delay distribution with bottleneck bandwidth of 100Mbps and varying number of TCP flows, $5ms \leq RTT \leq 10ms$

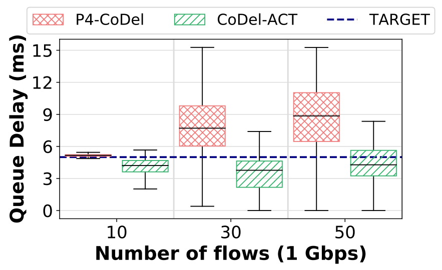

Figure 13. Queue Delay distribution with bottleneck bandwidth of 1Gbps and varying number of TCP flows, $3ms \leq RTT \leq 25ms$

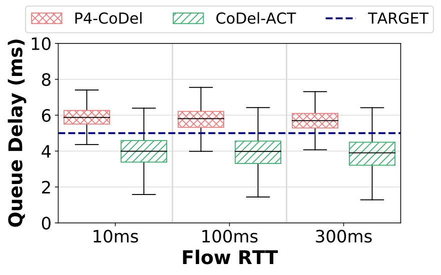

Figure 14. Queue Delay distribution with varying flow RTT, 3 TCP flows, and bottleneck bandwidth of 100Mbps

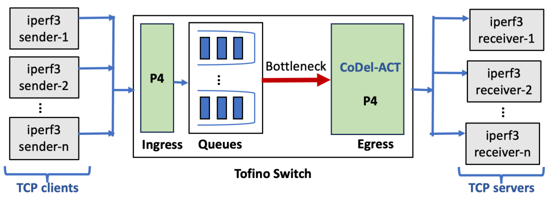

Figure 15. Experimental setup

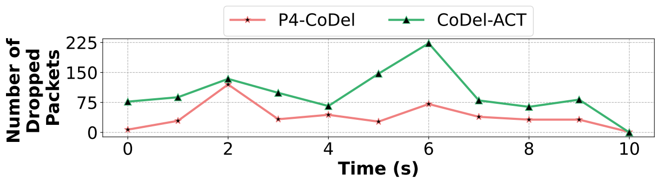

Figure 17. Number of packets dropped with bottleneck bandwidth of 100Mbps for 10 flows.

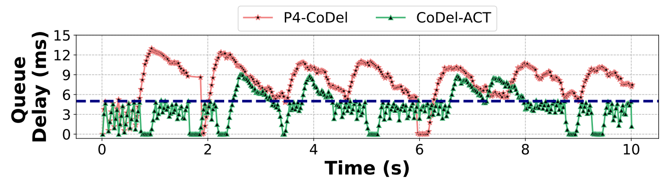

Figure 16. Queue Delay comparison with bottleneck bandwidth of 100Mbps for 10 TCP flows.

the TCP server (isolated using Linux namespaces). The total sending rate was capped at 90% of bottleneck bandwidth, and we used the Linux "tc" command to emulate each flow's RTT.

Intel Tofino configuration. In our evaluations, we compared *two* P4 programs for the Tofino switch: (1) `P4-CoDel`: an existing open source [26] CoDel implementation that runs in the egress pipeline, and (2) `CoDel-ACT`: a proposed CoDel implementation that we developed, which runs in the egress pipeline and leverages packet recirculation to synchronize register states across pipeline stages. To configure recirculation, we set up a mirroring session via the control plane.

For both programs, we integrated an L1 port bridging P4 program with the switch ingress pipeline. This program forwarded all packets from the TCP client to the TCP server. Additionally, we configured the egress port queue to shape traffic at either $100Mbps$ or $1Gbps$ (based on the experiment) and emulate congestion at the switch.

Parameters and metrics: We set CoDel parameters, $INTERVAL = 100ms$ and $TARGET = 5ms$, as per the RFC. We generate different workload scenarios by varying the number of parallel TCP flows, bottleneck bandwidth shared by the flows, and flow round trip time. All results reported are averaged over *three* runs of an experiment conducted for 10 seconds. The performance metrics measured are packet queueing delay and goodput. We have measured the algorithm's effectiveness (aggressiveness) in terms of the number of dropped packets, where a higher number is better. Additionally, we have measured the algorithm's design overheads in terms of packet recirculation bandwidth.

Research questions. We plan to answer the following questions through our evaluation. ① How does `CoDel-ACT` perform compared to existing solution, `P4-CoDel`? ② What makes `CoDel-ACT` aggressive compared to the existing solution? ③ What is the impact of packet recirculation on switch bandwidth utilization?

Varying number of flows, flow RTT ranges between 3ms to 25ms Fig. 12 shows the queue delay incurred by packets with `CoDel-ACT` and `P4-CoDel` for 1, 3, and 10 parallel TCP flows. We observe that `CoDel-ACT` always keeps the average queue delay well below the *TARGET*, whereas the average queue delay with `P4-CoDel` exceeded the *TARGET* by up to 43%.

Fig. 16 shows the observed packet queue delays for 10 flows with respect to time. The queue delay for `P4-CoDel` goes up to *12 ms* and remains high for over one second. However, `CoDel-ACT` exceeds the *TARGET* only for a brief period, and CoDel is triggered to initiate packet drops. This proves the effectiveness of `CoDel-ACT` by incorporating the *codel_init* function.

We observe a similar trend when the bottleneck bandwidth is set to 1 Gbps for 10, 30, and 50 TCP flows (see Fig. 13). However, we see that `P4-CoDel`'s performance degrades with an increase in bottleneck bandwidth. For both 30 and 50 TCP flows, the average queue delay exceeded the *TARGET* by up to 45%.

Varying flow RTT, bottleneck bandwidth=100Mbps, number of TCP flows=3. As per the RFC, CoDel provides excellent performance for flows with RTTs ranging from 10

Table I
GOODPUT (IN MILLION BITS PER SEC) FOR 10 FLOWS)

Bottleneck bandwidth	P4-CoDel	CoDel-ACT
$100Mbps$	82	82.4
$1Gbps$	832.6	842.3

ms to 300 ms [1]. In this experiment, we vary the flow RTT to 10ms, 100ms, and 300ms to observe if CoDel-ACT is sensitive to flow RTT. Fig. 14 shows that even with different flow RTTs, the CoDel-ACT's average queue delay was well below $TARGET$. Further, it outperforms P4-CoDel.

Testing CoDel-ACT's aggressiveness. Fig. 17 shows the number of dropped packets with 10 TCP flows. We can observe that CoDel-ACT drops significantly more packets than P4-CoDel. This indicates that *codel_init()*'s aggressiveness, present in CoDel-ACT but missing in P4-CoDel, is responsible for this difference. To investigate the impact of dropped packets, we compare the goodput of CoDel-ACT and P4-CoDel. Table I shows that the goodput achieved with CoDel-ACT and P4-CoDel is almost the same, with only a slight increase in goodput for CoDel-ACT, while both are very close to the sending rate of 90Mbps and 900Mbps.

Impact of packet recirculation and delayed state updates. Packet recirculation generates additional packets, which can result in a waste of network bandwidth and delayed state updates. Our design imposes a low packet recirculation rate (300 to 620 packets per sec) since we trigger the recirculation process only when the system enters or comes out of the congested queue cycles. Our evaluations show that the delayed state updates caused by the recirculation process do not significantly affect the effectiveness of our solution.

VII. CONCLUSION

We designed and implemented the RFC-compliant CoDel AQM algorithm for the Intel Tofino switch. CoDel-ACT reduces average queue delay by up to 52% compared to the existing Tofino solution, with the worst-case network bandwidth wastage of up to 4%. Some existing works combine functions such as traffic classification [27], scheduling [28], traffic prioritization [29], fairness [30], and QoS management [31] with CoDel for P4 Tofino switches. The CoDel algorithm in these works could be replaced by CoDel-ACT to improve their efficacy.

REFERENCES

[1] K. Nichols *et al.*, "Controlled delay active queue management," IETF, RFC 8289, 2018.

[2] J. Gettys and K. Nichols, "Bufferbloat: dark buffers in the internet," *Communications of the ACM*, vol. 55, no. 1, p. 57–65, 2012.

[3] 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.

[4] R. Pan *et al.*, "Proportional integral controller enhanced (pie): A lightweight control scheme to address the bufferbloat problem," IETF, RFC 8033, 2017.

[5] F. Schwarzkopf *et al.*, "Performance analysis of codel and pie for saturated tcp sources," in *28th International Teletraffic Congress (ITC 28)*, vol. 01, 2016.

[6] S. Laki *et al.*, "Towards an aqm evaluation testbed with p4 and dpdk," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, 2019.

[7] P. Vörös *et al.*, "T4p4s: A target-independent compiler for protocol-independent packet processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing*, 2018.

[8] M. Menth *et al.*, "Implementation and evaluation of activity-based congestion management using p4 (p4-abc)," *Future Internet*, vol. 11, no. 7, p. 159, 2019.

[9] C. Papagianni and K. De Schepper, "Pi2 for p4: An active queue management scheme for programmable data planes," in *Proceedings of the 15th International Conference on Emerging Networking Experiments and Technologies*, 2019.

[10] "Behavioral model (bmv2)," Available online at https://github.com/p4lang/behavioral-model.

[11] "Cisco catalyst 9000," Available online at https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-9000/white-paper-c11-742388.html.

[12] "Explicit congestion notification (ecn)," Available online at https://www.arista.com/en/um-eos/eos-quality-of-service#xx1166435.

[13] https://www-res.cablelabs.com/wp-content/uploads/2019/02/28094021/DOCSIS-AQM_May2014.pdf.

[14] T. Høiland-Jørgensen *et al.*, "Piece of cake: a comprehensive queue management solution for home gateways," in *IEEE LANMAN*, 2018, pp. 37–42.

[15] A. Sivaraman *et al.*, "No silver bullet: Extending sdn to the data plane," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, 2013.

[16] Intel tofino - intelligent fabric processors. Available online at https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

[17] R. Kundel *et al.*, "P4-codel: Experiences on programmable data plane hardware," in *IEEE International Conference on Communications*, 2021.

[18] X. Du, Xu *et al.*, "R-aqm: Reverse ack active queue management in multitenant data centers," *IEEE/ACM Transactions on Networking*, 2022.

[19] Agilio cx smartnics. Available online at https://www.netronome.com/products/agilio-cx/.

[20] M. Alizadeh *et al.*, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. Association for Computing Machinery, 2010, p. 63–74.

[21] X. Du *et al.*, "R-aqm: Reverse ack active queue management in multitenant data centers," *IEEE/ACM Transactions on Networking*, vol. 31, no. 2, pp. 526–541, 2023.

[22] H. Wu *et al.*, "Ictcp: Incast congestion control for tcp in data center networks," ser. Co-NEXT '10, 2010.

[23] F. Baker and G. Fairhurst, "Ietf recommendations regarding active queue management," https://datatracker.ietf.org/doc/draft-ietf-aqm-recommendation/09/, 2015.

[24] Peterson, Brakmo, and Davie, "Tcp congestion control: A systems approach," https://tcpcc.systemsapproach.org/aqm.html, 2022.

[25] M. Menth and S. Veith, "Active queue management based on congestion policing (cp-aqm)," in *International Conference on Measurement, Modelling and Evaluation of Computing Systems*. Springer, 2018.

[26] P4-codel. Available online at https://github.com/ralfkundel/p4-codel.

[27] Q. Wu and o. Liu, "P4sqa: A p4 switch-based qos assurance mechanism for sdn," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4875–4886, 2023.

[28] A. G. Alcoz *et al.*, "Everything matters in programmable packet scheduling," 2023.

[29] T. V. Doan *et al.*, "Interplay between priority queues and controlled delay in programmable data planes," in *2023 18th Wireless On-Demand Network Systems and Services Conference (WONS)*, 2023.

[30] W. G. de Morais *et al.*, "Application of active queue management for real-time adaptive video streaming," *Telecommunication Systems*, pp. 1–10, 2022.

[31] O. Lhamo *et al.*, "Red-sp-codel: Random early detection with static priority scheduling and controlled delay aqm in programmable data planes," *Computer Communications*, vol. 214, pp. 149–166, 2024.