USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Count-Based Abstractions for Performance Verification of Contention Points

Amir Seyhani, *University of Waterloo;* Aarti Gupta and David Walker, *Princeton University;* Mina Tahmasbi Arashloo, *University of Waterloo*

https://www.usenix.org/conference/nsdi26/presentation/seyhani

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

ARTIFACT EVALUATED USENIX AVAILABLE
ARTIFACT EVALUATED USENIX FUNCTIONAL
ARTIFACT EVALUATED USENIX REPRODUCED

Count-Based Abstractions for Performance Verification of Contention Points

Amir Seyhani
University of Waterloo

Aarti Gupta
Princeton University

David Walker
Princeton University

Mina Tahmasbi Arashloo
University of Waterloo

Abstract

Networking researchers and engineers spend much of their time trying to understand the performance of *contention points* – network components where multiple incoming packet streams share the same outgoing link(s). Recently, researchers have developed new logical models for analyzing such contention points, but unfortunately, such models are expensive: They do not scale well as buffer capacities increase beyond 10s of packets, making it difficult or impossible to reason about real-world systems faithfully. In this paper, we develop a suite of effective, new abstractions for reasoning about buffers and their performance characteristics. We also show how to architect a performance analysis framework for contention points in a modular way so it can take advantage of a range of abstractions that trade performance off against precision. We evaluate our abstractions against a collection of benchmarks and demonstrate their scaling benefits.

1 Introduction

Networking researchers and engineers spend much of their time trying to understand the performance of networked systems. Typically, they use simulation, emulation, or measurement of deployed systems to gain insight into how the network performs. However, one can only experiment with so many different concrete traffic traces or conditions before having to draw conclusions about the broader network performance properties – it is simply not possible to try all possible traces or conditions one at a time. Analytical frameworks like Network Calculus [1, 2] can derive bounds on metrics such as latency and throughput. However, these bounds are only informative when network behavior and traffic can be tightly approximated by concise, well-behaved algebraic functions, which limits their applicability to modern networks [3].

Symbolic reasoning offers a promising alternative for analyzing complex systems with diverse behaviors, as it can characterize all possible system behaviors at once, and naturally express a broader range of network functionality using logical formulas. Recent work, in particular FPerf [4] and CCAC [5], has shown promising results in applying symbolic reasoning to network performance analysis. They encode the semantics of network components and properties of interest as formulas and use SMT solvers such as Z3 [6] to verify them. CCAC focuses on modeling the behavior of congestion control algorithms over Internet paths, while FPerf focuses on analyzing a composition of *contention points*.

A contention point is any network component where multiple incoming packet streams share the same outgoing link(s), and the input rate may temporarily exceed the output rate. Contention points typically include First-In-First-Out (FIFO) buffers to absorb such transient bursts. When buffers do not drain as fast as expected, queues build up beyond acceptable thresholds, and significant performance problems arise. To diagnose and prevent such issues in advance, engineers need answers to key performance questions about buffer drain rate, buffer build-up, and packet drops. Moreover, since contention points mediate between input packet streams for access to the shared output(s), engineers also need to determine fairness properties and prioritization criteria for such systems.

In reasoning about contention points, FPerf uses a precise encoding for the central component – *the buffers*. For a buffer with capacity C whose behavior is modeled over T time steps, FPerf defines $C \times T$ sets of variables, each representing one element of the buffer at one timestep. Unfortunately, this encoding is extremely costly as the number of variables and constraints grows with buffer capacity (C), and SMT-solving scales exponentially in the worst case. As such, while FPerf can analyze examples with small buffer capacities (e.g., 10 packets), it does not scale up to buffer capacities in real networks, such as 100s of packets in wide-area-network switches and 1K to 10K packets in Linux QDiscs [7–9]. Indeed, for a system with 30 buffers, FPerf's verification goes from a few seconds for buffer capacities of 10 packets to over 200 seconds for buffer capacities of just 50 packets (§6.2).

Limiting analysis to small buffer capacities alters packet drop and delay behavior. The same input workload can exhibit different drop patterns and different queuing delays under different buffer sizes, including drops that would not occur in practice. Consequently, analysis with unrealistically small buffers can misrepresent real network behavior. Accurately modeling contention points, therefore, requires reasoning about buffers at realistic capacities.

One way to attack such a problem is to introduce abstractions that intentionally discard information during the modeling process for the sake of generating smaller, more tractable representations. In this case, because our goal is verification of contention point properties, such abstractions should result

in a *sound* over-approximation of contention point behavior. Of course, any such abstraction risks giving up too much accuracy to achieve the goal at hand – a useful abstraction must also be precise enough to be able to carry out the verification task. Finding the balance between verification cost and model accuracy is the main challenge in designing such abstractions.

In this paper, we develop new abstractions for buffers, the central component of any contention point. These abstractions use two key ideas. First, buffer contents are approximated using a vector of counts. Here, each element of the vector represents the number of packets of each traffic class of interest in the network (e.g., high, medium, and low priority traffic classes). That is, if there are K traffic classes one is interested in reasoning about, our abstraction maintains $K \times T$ variables for each buffer. K is not dependent on the buffer capacity. As such, buffer capacity will not significantly impact the verification time under this abstraction.

While extremely efficient, on its own, such an abstraction sheds too much packet ordering information among traffic classes as they flow through a contention point. As such, when packets from different traffic classes can get mixed within a buffer, we layer a second idea on top of the first by introducing windows over the sequence of packets entering and exiting that buffer. Within each window, a vector of counts tracks the number of packets from different traffic classes, ignoring packet ordering within that window. However, we enforce ordering across windows. This gives rise to a flexible abstraction to effectively trade precision for performance – smaller window sizes capture more precise ordering information across packets at the cost of higher verification time.

Which buffer representation to use, e.g., FPerf's fully precise model or our abstract representation, depends on the contention point and performance properties of interest. For smaller buffer capacities (10s of packets), a fully "spelled-out" buffer representation may perform better as there can be overheads associated with integer arithmetic for tracking counts. For larger buffer capacities and when packets from different traffic classes do not mix in buffers (e.g., stand-alone packet schedulers), we can avoid using windows and their associated overhead. Otherwise, using our full abstraction (counts + windows) would be the way to go.

To that end, we model contention points in a *modular* way, providing a light-weight, sufficient interface based on buffer backlogs for the contention point (mediation) logic to interact with buffers. This interface is inspired by recent work that suggests the ability to check if a buffer is backlogged, together with enqueue, dequeue, and a few other operations, is a promising abstract interface for buffers in contention points [10]. Our modular architecture implements such an interface, enabling users to swap in and out different buffer models at varying levels of abstraction depending on the use case, and making contention point logic amenable to being derived from higher-level programs in languages like Buffy [10].

Contributions. We make the following key contributions:

- We introduce vector of counts and window abstractions as mechanisms to represent buffers and control the cost-precision tradeoff in contention point verification. We give advice on choosing the right window size to benefit from cost reductions while retaining sufficient accuracy (§3).
- We show how we model the contention point's logic as a state machine that interacts with buffers through a minimal yet sufficient interface, and discuss its benefits (§4).
- We describe how we reason about a composition of contention points using our abstraction, and the impact of our over-approximation on the analysis results (§5).

We evaluate the effectiveness of our abstractions using a collection of benchmarks, demonstrating their scaling benefits over existing work within our modular model. We also show that our buffer interface aligns with the Buffy language [10], streamlining the translation of high-level programs that describe contention-point logic into our modular model (§6).

2 Key Ideas

In this section, after defining basic terminology, we describe our framework's key ideas: our vector count and window abstractions, and a modular contention-point architecture. In this paper, we use *buffer size and capacity interchangeably*.

2.1 Definitions and terminology

Contention points and buffers. A *contention point* is a shared resource that multiple packet streams attempt to access. Because the packet processing rate may not always keep up with arrival rate, network components buffer packets in temporary storage for later processing. We model a contention point as a processing unit that reads packets from several input buffers and produces several output streams (Figure 2)[1]. At each (discrete, abstract) timestep, a contention point decides, based on the status of the input buffers and its internal state, which buffers to dequeue from. It then moves packets from the input buffers to the output streams accordingly. Two contention points can be composed by directing the output stream of one to the input buffers of another. We assume FIFO buffers, the most common case in networked systems.

Performance metric sequences. Our goal is to analyze the performance of a system of contention points, composed as a directed acyclic graph. A *performance metric sequence* tracks the value of a buffer metric over a finite, discrete time horizon. Common, time-varying metrics of interest include, but are not limited to *input count* (packets arriving at the buffer), *output count* (departing packets), *buffer occupancy* (number of packets in the buffer), *drop count* (dropped packets), and *inter-packet gap* (the time since the previous arrival, a proxy for arrival rate). Similar performance metrics can be defined over bytes rather than packets. Throughout the paper, we use packets for ease of presentation.

[1]This model differs slightly from FPerf, where output is modeled as a set of buffers rather than packet streams

Performance properties. A *performance property* over a particular metric and buffer is a set of performance metric sequences for that buffer. For example, consider a performance property specifying that a given buffer should experience fewer than 5 packet drops over 10 timesteps. Formally, this property is expressed as the subset of all drop-count sequences $d_1, \cdots, d_{10}$ such that the sum is less than 5, i.e., $\Sigma_{i=1}^{10} d_i < 5$.

Workloads and queries. Consider a composition of contention points, e.g., two priority schedulers each feeding into one input buffer of a round-robin scheduler. We define "free" input buffers as those not connected to another contention point, e.g., the input buffers of the two priority schedulers. *Workloads* are performance properties over the free input buffers. Conceptually, they characterize the external traffic entering the system of contention points. *Queries* are performance properties over any buffer in the system and characterize the performance question we wish to analyze. The packet drops property described above is an example of a query.

2.2 Abstracting buffers

FPerf analyzes contention points with respect to queries describing undesirable performance properties over buffers. It uses syntax-guided synthesis (SyGuS) [11] to search for compact representations of input traces that lead to performance problems. This representation is in the form of properties over input buffers (i.e., workloads). During its search, FPerf makes *repeated* SMT solver calls to check if its candidate workload makes the following formula unsatisfiable:

$$P \wedge CP(I,O) \wedge \neg Q \tag{1}$$

Here, P is the logical formula encoding the workload (a property of the inputs), Q encodes the query, and CP encodes the contention point's packet-processing behavior in terms of input (I) and output (O) traffic streams. In other words, FPerf finds a workload P that violates the query Q.

FPerf does not scale to realistic buffer sizes. FPerf encodes buffers in an exact and precise manner – each buffer is modeled as a fixed-size array of packets, and each packet in the buffer is encoded with a separate set of individual variables. For example, consider a buffer with capacity 10, where each packet has two metadata fields of interest (e.g., traffic class and flow ID). Each buffer element is encoded using three variables, two for metadata and one boolean indicating whether the slot is occupied. Hence, we wind up with 30 variables for one 10-element buffer. Then, a set of SMT formulas (included in CP above) models how these variables change under operations such as enqueue and dequeue.

Figure 1 shows how FPerf's verification time increases with the buffer size when checking whether a given workload satisfies a query (equation 2). FPerf performs 100s of such calls during its workload search. Increasing buffer size results in larger verification times, since the number of variables increases linearly *and the search space grows exponentially*.

In fact, FPerf's case studies all use buffers of capacity 10,

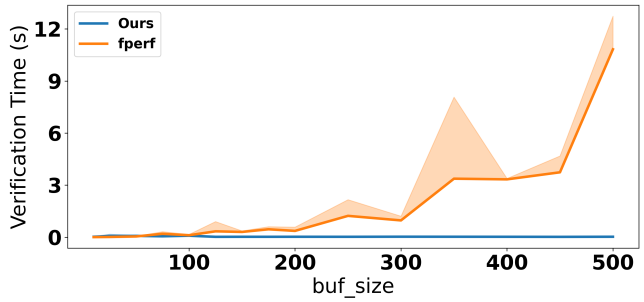


Figure 1: Average (line) and P95 (shaded) verification time for a strict priority scheduler under FPerf and our approach.

far smaller than those in real networks. For example, default queue sizes in Linux QDiscs [12] range from 1K to 10K packets [9], and switches, particularly in wide-area networks, can have per-port buffers with hundreds of packets [7, 8]. Even for simple case studies such as a priority scheduler, FPerf's scalability degrades beyond queue sizes of 100 packets. This threshold is even lower for more complex contention points (§6.2). Consequently, while FPerf provides useful intuition from small-scale experiments, it cannot analyze systems with realistic buffer sizes. This limitation is fundamental: buffer size is a critical contention point parameter that directly affects performance metrics such as packet drops and delay. That is, under a smaller-than-realistic buffer size, the same input workload can exhibit different drop patterns, including drops that would not occur in practice, and different queuing delays. Thus, conclusions drawn from unrealistically small buffers may mischaracterize network behavior.

A single count for each buffer. Our first step towards a more scalable buffer encoding is to model each buffer as a single integer. That is, instead of modeling individual packets, we represent the state of each buffer at each timestep with a single integer that represents the number of packets it contains at that timestep. As such, our encoding size for a contention point becomes independent of its buffer capacities, and verification effort will scale independently of buffer size.

This abstraction is simple yet powerful. To see how it works, consider a strict-priority scheduler with four input buffers and one output stream, each with capacity five, as the contention point. At each timestep, the scheduler dequeues one packet from the highest-priority *backlogged*, i.e., non-empty, buffer. Let workload P specify that in each timestep up to time 5, at least one packet is enqueued into buffer 1 or 2. Let query Q specify that in these 5 timesteps, no packet from buffer 3 reaches the output stream. If the verifier returns UNSAT for $P \wedge CP(I,O) \wedge \neg Q$, where CP is the SMT encoding of the scheduler, then as long as the input traffic satisfies the property P, no packet from buffer 3 enters the output stream.

In this example, tracking packet counts alone suffices to model the contention point behavior and reason about the query. That is because the query depends only on how many packets are dequeued from buffer 3, and the contention point's

decisions depend only on whether each buffer is backlogged or not. Therefore, individual packets and their content are irrelevant for this analysis, so modeling them explicitly is unnecessary. Our experimental results show that this abstraction is scalable with respect to buffer capacity, as verification time does not grow with increasing buffer size (§6).

A vector of counts for each buffer. While promising, a single packet count per buffer is not sufficient for modeling a diverse set of scenarios, as it cannot differentiate between packets from different traffic classes. Consider the priority scheduler from the previous example, but with two input buffers. Packets in the first buffer come from a "high-priority" application and those in the second buffer from a "low-priority" one, both belonging to tenant A in a cloud environment. Suppose a similar priority scheduler exists for tenant B, and the outputs of both schedulers feed into a round-robin scheduler that alternates between A and B. Suppose we wish to reason about the difference between the number of high- and low-priority packets in the round-robin output. A single count per buffer is insufficient as it only tracks the total number of packets in each buffer, losing information about how many are high- and low-priority.

As such, we generalize our abstraction to represent each buffer as a *vector of packet counts*. The vector size is the number of traffic classes of interest, and each vector element tracks the number of packets for each traffic class in the buffer. The encoding size remains independent of buffer capacity, making verification time scalable with respect to buffer size.

Counts are not enough. By abstracting the buffer state as a vector of packet counts, we lose information about the order in which packets arrive at and are stored in the buffer. To see why this matters, consider the toy example of a round robin scheduler feeding a simple rate limiter. The scheduler has two input buffers: one only receives packets from traffic class A and the other only from class B. Its output stream, containing a mix of A and B packets, feeds the rate limiter, which transmits one packet every other timestep from its input buffer of size 4. Suppose the workload specifies that one A and one B packet arrive at the round-robin scheduler each time step, and the query asks if four consecutive B packets can appear in the rate limiter's output over 10 timesteps.

Clearly, this workload does not imply the query: since both round-robin input buffers are always backlogged, its output, and hence the rate limiter's input and output, should alternate between A and B packets. However, a naive count-based abstraction would conclude otherwise. Let $\langle a, b\rangle$ denote the state of the rate limiter's input buffer, where a and b are the numbers of A and B packets. Every two timesteps, one A and one B packet enter the buffer and one departs. Without ordering constraints, when the rate limiter requests a dequeue from its input buffer, the abstract buffer model allows returning any packet consistent with the buffer's packet counts. Suppose the abstract buffer returns an A packet on the first dequeue and B packets thereafter, with dequeues occurring on even timesteps.

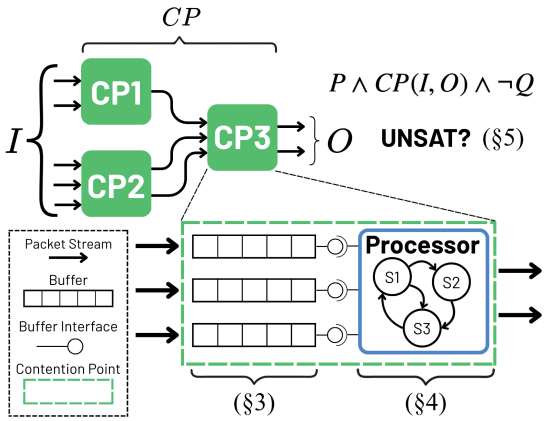


Figure 2: High-level contention-point model

Then, the buffer state evolves as $\langle 0, 1\rangle$ after timestep 2, $\langle 1, 1\rangle$ after timestep 4, $\langle 2, 1\rangle$ after timestep 6, and so on. Continuing this pattern results in four consecutive B packets in the rate limiter's output stream, spuriously satisfying the query. In fact, an unconstrained count-based abstraction permits outputs consisting only of B packets with A packets never leaving the buffer after timestep 2

Our proposal: counts + windows (§3). Without ordering constraints, a count-based abstraction over-approximates buffer behavior and becomes too imprecise. Conversely, precisely modeling the ordering between all packets, as FPerf does, does not scale to realistic buffer sizes. We propose to add ordering constraints over *windows* of packets instead. That is, we group packets entering and leaving the buffer into fixed-size windows, and enforce ordering across windows, but not within them.

In our running example, suppose the window size is four. Our constraints enforce that packet counts match between corresponding input and output windows (e.g., input and output packets 1–4, 5–8, and so on). After the first four packets enter the queue (end of timestep 4), the first input window has counts $\langle 2, 2\rangle$, the first output window has $\langle 1, 1\rangle$, and the buffer state is $\langle 1, 1\rangle$. Suppose we dequeue a B packet in timestep 6. The buffer state becomes $\langle 2, 1\rangle$, and the first output window becomes $\langle 1, 2\rangle$. At this point, unless an A packet is dequeued, the first input and output windows will not match. So, the next packet dequeued at timestep 8 must be an A.

The window constraints do not ensure perfect ordering – the B packet dequeued at time 6 arrived after the A packet dequeued at time 8. But, the window constraints ensure that an A packet does not depart more than W packets after it arrives, where W is the window size. The smaller the window, the more "precise" the window constraints. In fact, for a window size of one, our buffer model would be equivalent to FPerf's, though of course it would be just as costly as FPerf's as well. Overall, this window-and-counts abstraction introduces an effective, new, and general *pay-for-what-you-use* mechanism for trading off precision for performance. In addition to introducing a general new abstraction, we give advice on setting

window sizes in practical settings (§3.2). In particular, we argue for setting the window size equal to buffer capacity as the right middle ground for enforcing sufficient ordering and remaining scalable. Our experimental results (Figure 1) show that our counts-and-windows abstract buffer model outperforms FPerf and scales to realistic buffer capacities.

2.3 A modular contention-point architecture

We model a contention point as a *processing unit* that reads packets from several input buffers and produces multiple streams of output traffic. The processing unit represents the contention-point logic that mediates between contending traffic stored in the input buffers. This logic can implement a packet scheduler, a switch/router, or even a congestion control algorithm [10]. At every time step, it decides, based on buffer states and its own internal state, which buffers send how many packets and to which output streams.

We observe that the information the processing logic needs from input buffers to make such decisions can be captured in a light-weight, well-defined interface. Specifically, it does not need to know individual packet ordering in a buffer to decide if that buffer gets to transmit. In fact, in most practical cases, decisions depend only on buffer backlogs. For example, the fair queuing logic of FQ-CoDel maintains two lists of new and long-standing backlogged buffers, and services each list in a round-robin fashion, prioritizing the first one [4,13]. Similarly, a switch crossbar using iSLIP matches input buffers (virtual output queues) to output streams based on the matching in the previous timestep (its own internal state) and current buffer backlogs [14]. In Deficit Round-Robin [15], widely deployed in switches and routers, the scheduler tracks per-buffer deficits (its own internal state) and serves backlogged buffers in round-robin order and based on their deficit.

Given this observation, our logical model of a contention point (Figure 2) has three components: (1) a model of the *input buffers* (§3), (2) A model of the processing logic, encoded as a *state machine* controlling the contention point's decision making (§4), and (3) an *interface*, based on buffer backlogs, that glues the two together. Specifically, at each time step, buffers share whether they are backlogged, and optionally their backlog size, and the processing logic decides how many packets to dequeue from each. Our count-based buffers fit quite well with this interface – they preserve the backlog information needed for the processing logic's decision making while abstracting away fine-grained packet ordering.

Separating contention point logic from buffer representation through a consistent interface has several advantages (§4). First, it allows users to swap buffer models at different abstraction levels depending on their use case. They can use a more precise model, like FPerf's, if their buffer sizes are small and they can afford its high verification cost, or our abstraction with their desired window size if they want to trade off precision for performance. Second, buffer representations and optimizations can more easily be reused across different contention points, as opposed to being implemented from scratch. Third, it enables generating the processing logic from higher-level programs written in languages like Buffy [10].

3 Buffer as a Vector of Counts

Our buffer model includes (1) a representation of the buffer state, maintained in separate variables for each time step, (2) logic controlling the way packets are enqueued at each timestep, and (3) logic governing packet dequeues at each time step. Contention-point implementations are typically deterministic:: Given a sequence of inputs, they produce the same outputs. Our abstract model is non-deterministic: By reducing the amount of logic used to represent buffers, we represent the contention point behavior more efficiently, but less exactly. Our abstract model represents many behaviors simultaneously, but as long as it includes the true behaviors, the abstraction is sound for verification.

Notation. We use bold font ($\mathbf{x}$) to denote a vector of objects of type x. For a vector of integers $\mathbf{x}$, $\sigma(\mathbf{x})$ is the sum of its elements. See §A for the semantics of vector operations.

3.1 The Base Abstraction

(1) A sequence of packet count vectors. We model a buffer as a vector of counts updated at each timestep. The vector size is the number of traffic classes of interest, and the ith vector element represents the number of packets of class i in the buffer. More formally, with K traffic classes, a *packet count vector* is defined as $\boldsymbol{v} = \langle v_1, \ldots, v_K \rangle$, where $v_k \in \mathbb{N}$ represents packet counts of the k-th class. To model a buffer over T time steps, we use a sequence of packet count vectors $\boldsymbol{v}_1, \cdots, \boldsymbol{v}_T$, where $\boldsymbol{v}_t$ is the buffer state at time t. The cost of such a representation is on the order of $N \cdot K \cdot T$ where N is the cost of representing an individual count. Figure 3 shows a concrete buffer and its abstraction as a vector of counts, and how they get updated in response to enqueues and dequeues within a timestep (enclosed in the dashed box).

(2) Logic for enqueue (and drops). At the start of timestep t, the buffer state reflects packets carried over from timestep $t-1$. We first handle enqueues. In the concrete buffer model, the sequence of packets arriving at the buffer right before timestep t is appended to it in order until it is full. The rest are dropped. For example, in Figure 3, the buffer has capacity four and holds three packets. Thus, out of the three arriving packets, the first is enqueued and the last two are dropped.

In the abstract model, arrivals are modeled as a vector of counts, capturing the per-traffic-class packet counts arriving right before t. This abstracts away packet arrival ordering within a timestep. So, if there is insufficient buffer capacity, the buffer model nondeterministically chooses how many packets of each class to enqueue and drop, and updates the abstract buffer state accordingly. The top part of Figure 4 illustrates the non-deterministic enqueues and drops. Let $\boldsymbol{i}_t$ denote the packet count vector for arrivals at the buffer right before t, $\boldsymbol{e}_t$ the packet count vector for those enqueued at time t, and

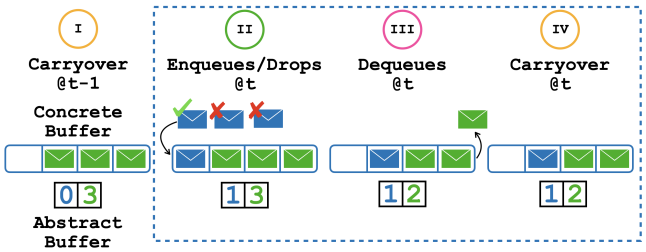


Figure 3: Modeling a buffer as a vector of counts

$\boldsymbol{d}_t$ the packet count vector for those dropped at time t. Let b_t be a boolean indicating whether the buffer is backlogged after enqueues (true if the buffer is not empty). The following constraints capture this non-deterministic choice:

$$EnqDropSum(\boldsymbol{i}_t, \boldsymbol{e}_t, \boldsymbol{d}_t) = \boldsymbol{i}_t = \boldsymbol{e}_t + \boldsymbol{d}_t$$

$$Backlog(b_t, \boldsymbol{e}_t, \boldsymbol{v}_{t-1}) = b_t \iff \boldsymbol{v}_{t-1} + \boldsymbol{e}_t > \boldsymbol{0}$$

$$Drop(\boldsymbol{d}_t, b_t, \boldsymbol{v}_i, \boldsymbol{v}_{t-1}) = \begin{cases} \boldsymbol{d}_t > \boldsymbol{0} \Rightarrow \sigma(\boldsymbol{v}_{t-1} + \boldsymbol{e}_t) = C & \text{if } b_t \\ \boldsymbol{d}_t = \boldsymbol{0} & \text{else} \end{cases}$$

where C is the buffer capacity. *EnqDropSum* simply enforces that arrivals equal the sum of enqueued and dropped packets. *Backlog* defines backlog b_t after enqueues and before dequeues. The backlog variables are part of the interface between the buffers and the processing logic. They are used by the state machine encoding how the contention point mediates between buffers to make its decisions (§4). *Drop* ensures drops only happen when a buffer is backlogged and full.

(3) Logic for dequeue. Next, the processing logic uses backlog information (b_t) from all buffers to decide how many packets to dequeue from each. In Figure 3, suppose the processing logic decides to dequeue one packet from this buffer at time t. A concrete buffer model would track the ordering of individual packets within the buffer. So, it will dequeue the green packet at the buffer head, and leave the two green and one blue packet. Our abstract model does not track such ordering. As such, it nondeterministically selects a packet from any class with available packets (subject to window constraints; §3.2), as shown in the bottom of Figure 4. Let m_t be the number of packets the processing logic dequeues from this buffer and $\boldsymbol{o}_t$ the packet count vector representing how many packets of each class are dequeued, at time t. The *Deq* constraint updates the buffer state and ensures that the sum (σ) of packet counts in $\boldsymbol{o}_t$ matches m_t:

$$Deq(\boldsymbol{e}_t, \boldsymbol{o}_t, m_t, \boldsymbol{v}_{t-1}, \boldsymbol{v}_t) = (\boldsymbol{v}_t = \boldsymbol{v}_{t-1} + \boldsymbol{e}_t - \boldsymbol{o}_t) \wedge \sigma(\boldsymbol{o}_t) = m_t$$

3.2 Balancing Precision and Performance

The abstract buffer model described so far over-approximates the behavior of a FIFO buffer. To see why, let $[a_1, a_2, \cdots]$ denote the sequence of packets enqueued into a buffer, where a_i is the traffic class of the ith packet ($1 \le a_i \le K$). For simplicity, we assume at most one packet is enqueued in each timestep. This is not a "timed" sequence. That is, the subscripts capture

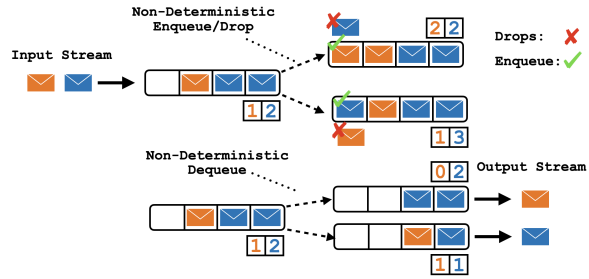


Figure 4: Non-Deterministic Enqueuing and Dequeuing

only enqueue order, not enqueue time. For example, a_1 was enqueued before a_2, but the two packets could have been enqueued in consecutive timesteps or farther apart. Similarly, let $[r_1, r_2, \cdots]$ denote the departure sequence.

In a concrete model of a FIFO buffer, packets depart in the same order they are enqueued ($r_i = a_i$). The count-based abstraction, however, loses information about packet enqueue orders and may allow departures not fully aligned with the enqueue sequence. For example, suppose packets $[b, c, c]$ are enqueued at $t = 1$ and $[c, c]$ at $t = 2$, resulting in the enqueue sequence $[b, c, c, c, c]$. After the enqueues at $t = 1$, the abstract buffer state is $\langle 1, 2 \rangle$ (one b, two cs). If the contention point dequeues one packet per timestep, the model may choose to output a c packet, even though a b packet was enqueued earlier, without contradicting any of its constraints. Repeating this choice results in the dequeue sequence $[c, c, c, c, b]$, which is consistent with the counts but not with FIFO ordering.

Without additional constraints, as long as the buffer stays backlogged, a packet may appear arbitrarily far from its enqueue position in the departure sequence. This is not always problematic. To see why, note that packets are distinguished only at the granularity of traffic classes. That is, packets within the same class are interchangeable and their relative order is irrelevant to the analysis. So, the above problem will not happen if a buffer carries packets from a single traffic class. It happens only when packets from multiple traffic classes coexist in a buffer. In such cases, we must bound the reordering between packets of different classes in the departure sequence to obtain a meaningful approximation of FIFO behavior.

Bounding re-ordering with windows. Our abstract buffer model bounds re-ordering by grouping enqueued and dequeued packets into non-overlapping windows of fixed size, and enforcing order at the granularity of windows. Intuitively, for a window size of W, our model enforces that the first W packets of the enqueue sequence ($[a_1, a_2, \cdots, a_W]$) and the first W packets of the dequeue sequence ($[r_1, r_2, \cdots, r_W]$) contain the same number of packets from each traffic class. It ensures this property for all subsequent non-overlapping windows as well. That is, per-class packet counts will match in $[a_{W+1}, \cdots, a_{2W}]$ and $[r_{W+1}, \cdots, r_{2W}]$, $[a_{2W+1}, \cdots, a_{3W}]$ and $[r_{2W+1}, \cdots, r_{3W}]$, and so on. This bounds the extent of reordering relative to FIFO behavior to the window size W.

Figure 5 illustrates this mechanism for one buffer over

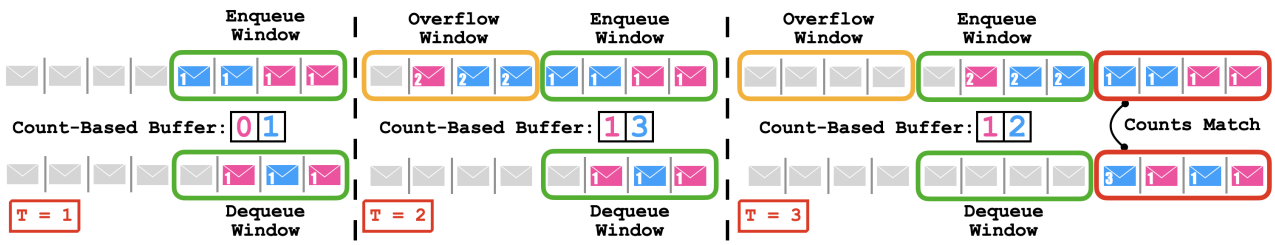


Figure 5: Bounding re-ordering with windows. **Legend:** enqueue (top) and dequeue (bottom) sequences with relevant packets labeled with enqueue and departure times. The abstract buffer (middle) maintains per-class packet counts. **T=1:** three packets depart, one blue packet left in the buffer. **T=2:** three new packets arrive. Enqueue and dequeue windows do not yet match. Packets departed at T=1 remain tracked in the enqueue window. New arrivals are tracked in the overflow window. **T=3:** One more dequeue. The counts in enqueue and dequeue windows match. They are flushed and the overflow replaces the enqueue window.

three timesteps, $W = 4$, and traffic classes pink and blue. Intuitively, our goal is to enforce per-class consistency between enqueues and dequeues in batches of four packets. At $t = 1$, the enqueue window records two blue and two pink enqueues, and the dequeue window records one blue and two pink departures, leaving one blue packet in the buffer. At $t = 2$, one pink and two blue packets are enqueued. Since the enqueue window is full and does not yet match the dequeue window, the new packets are recorded in a separate *overflow window*. The buffer now holds one pink and three blue packets. At $t = 3$, a blue packet is dequeued, causing the enqueue and dequeue windows to match (red borders). This indicates that, for the first four packets, the per-class numbers of enqueues and dequeues are equal. Both windows can then be flushed (green border) to begin tracking the next batch, and packets in the overflow window are recorded in the new enqueue window.

Navigating window size trade-offs. The count-based abstraction with window constraints provides an effective mechanism to trade off precision for performance. For a window size W, our model would need to maintain a sequence of $N = \lceil \frac{C}{W} \rceil + 1$ enqueue windows $\boldsymbol{w}^{e_1}, \cdots, \boldsymbol{w}^{e_N}$, and a dequeue window $\boldsymbol{w}^o$, where C is the buffer capacity and each window is a packet count vector. Initially, $\boldsymbol{w}^{e_1}$ tracks per-class counts for the first W enqueued packets and is updated as new packets are enqueued. As packets depart, $\boldsymbol{w}^o$ is updated to maintain per-class packet counts for the most recent W departures.

When the counts in $\boldsymbol{w}^{e_1}$ and $\boldsymbol{w}^o$ match, the first window of packets has fully departed. $\boldsymbol{w}^o$ resets to all zeros, and the enqueue windows shift forward so that $\boldsymbol{w}^{e_1}$ now tracks the earliest window still in the buffer (the second W packets in the entire enqueue sequence). Similarly, $\boldsymbol{w}^{e_i}$ maintains per-class packet counts for the ith window of enqueued packets that have not fully left the buffer yet, and is updated accordingly as earlier windows of packets leave the buffer. Note that we do not enforce the order in which packets within a window leave the buffer – we ensure all packets in one window depart before those in the next. Later in this section, we formalize these constraints and address corner cases.

Why only $N = \lceil \frac{C}{W} \rceil + 1$ enqueue windows? At any point in time, we need to track at most $C + W$ packets: the C packets currently in the buffer, plus the W packets from the first enqueue window that has not yet fully departed (some packets already dequeued, some still buffered). Smaller windows increase N, resulting in more variables and constraints and higher verification cost. In the extreme case $W = 1$, our buffer model would track $C + 1$ windows of size one, reducing to FPerf's buffer representation and enforcing exact FIFO ordering. Larger windows reduce N and improve scalability, but reduce precision as ordering is not enforced within a window.

Picking the window size. While our abstraction allows any window size that best fits the precision and performance needs of individual use cases, we argue for setting W equal to the buffer capacity C. The intuition is simple: buffers exist to absorb bursts, and their capacity corresponds to an (conservative) estimate of the expected burst size. Within a burst, packets arrive close together in time, and small variations in upstream delays can already change their relative order. Setting $W = C$ relaxes ordering within bursts, where fine-grained ordering is often not semantically meaningful, while preserving ordering across bursts. Moreover, this requires tracking only two enqueue windows, reducing model complexity.

Formalizing window constraints. Recall that $\boldsymbol{e}_t$ and $\boldsymbol{o}_t$ are packet count vectors representing per-class counts of enqueued and dequeued packets at time t. Assuming $W = C$, we use $\boldsymbol{w}^e_t$ and $\boldsymbol{w}^n_t$ to track per-class counts for the first and second (overflow) enqueue windows at time t, and $\boldsymbol{w}^o_t$ to track counts in the dequeue window. We update the windows from time $t-1$ to t in two steps, using three packet count vectors $\boldsymbol{x}^e_t, \boldsymbol{x}^n_t$, and $\boldsymbol{x}^o_t$ to capture the intermediate state of the windows between these steps.

First, we determine whether to add the newly enqueued packets ($\boldsymbol{e}_t$) to the first window ($\boldsymbol{w}^e_{t-1}$) or the second ($\boldsymbol{w}^n_{t-1}$). If the first window already tracks p packets and the remaining window capacity is insufficient for the new packets, the model nondeterministically assigns $W - p$ packets to the first window and the rest to the second:

$$
\begin{aligned}
&[\boldsymbol{w}^e_{t-1} + \boldsymbol{w}^n_{t-1} + \boldsymbol{e}_t = \boldsymbol{x}^e_t + \boldsymbol{x}^n_t] \\
&\wedge [\sigma(\boldsymbol{x}^n_t) > 0 \Rightarrow \sigma(\boldsymbol{x}^e_t) = C] \wedge [\sigma(\boldsymbol{x}^e_t) < C \Rightarrow \sigma(\boldsymbol{x}^n_t) = 0] \\
&\wedge [\neg(\sigma(\boldsymbol{x}^e_t) < C \wedge \sigma(\boldsymbol{x}^n_t) > 0)] \wedge [\boldsymbol{w}^e_{t-1} \leq \boldsymbol{x}^e_t]
\end{aligned}
$$

The first constraint ensures that all packets from the previous enqueue windows *and* the new enqueues are assigned

to either the first or second intermediate windows. The final constraint ensures that packets already in $\boldsymbol{w}^e_{t-1}$ remain in the first intermediate window. The remaining constraints ensure that packets appear in the second intermediate window only after the first one is full. We also update the (intermediate) dequeue window: $\boldsymbol{x}^o_t = \boldsymbol{w}^o_{t-1} + \boldsymbol{o}_t$.

Second, we check if the packet counts in the first enqueue window and the dequeue window match. If so, we reset the dequeue window, replace the first enqueue window with the second, and clear the second enqueue window. The enqueue and dequeue windows do not need to be full to match. Whenever the counts match, we can "opportunistically" move on:

$$\boldsymbol{w}^e_t = \begin{cases} \boldsymbol{w}^n_{t-1} + \boldsymbol{x}^n_t, & \boldsymbol{x}^e_t \leq \boldsymbol{x}^o_t \\ \boldsymbol{x}^e_t & \text{else} \end{cases} \wedge \boldsymbol{w}^n_t = \begin{cases} \boldsymbol{0}, & \boldsymbol{x}^e_t \leq \boldsymbol{x}^o_t \\ \boldsymbol{w}^n_{t-1} + \boldsymbol{x}^n_t & \text{else} \end{cases}$$

$$\boldsymbol{w}^o_t = \begin{cases} \boldsymbol{x}^o_t - \boldsymbol{x}^e_t, & \boldsymbol{x}^e_t \leq \boldsymbol{x}^o_t \\ \boldsymbol{x}^o_t & \text{else} \end{cases} \wedge \boldsymbol{w}^o_t \leq \boldsymbol{w}^e_t$$

4 Processing logic as a state machine

We model a contention point's processing logic as a state machine. The states typically record information about (1) buffer backlogs, and (2) past decisions, such as which buffers were recently serviced and to what extent. In addition, each state is associated with an *action*, which determines how many packets should be dequeued from each input buffer at that timestep. The machine transitions from one state to the next at each timestep, based on updated backlog information obtained through the buffer interface.

Connecting buffers and processing logic. Recall from §3 that $\boldsymbol{e}_t$ is the packet count vector representing the packets enqueued into the buffer at time t. The *Backlog* constraint updates b_t to reflect whether the buffer is empty after enqueues and before dequeues. The backlog variables b_t, and optionally buffer sizes (after enqueues and before dequeues) $\sigma(\boldsymbol{v}_{t-1} + \boldsymbol{e}_t)$, from all the contention points' buffers are shared with the processing logic, and can be used to make dequeue decisions. The processing logic then determines the m_t for each buffer, determining how many packets should be dequeued. The buffer state is then updated accordingly (§3).

Formally, we model a contention point as:

$$CP(I,O) = \exists B, \bigwedge_{i=1}^{N} Buff(I^i, B^i, M^i, L^i) \wedge \\ Process(B,M) \wedge Out(L,M,O)$$

Buff denotes the buffer constraints from §3, i.e., the conjunction of *EnqDropSum*, *Backlog*, *Drop*, *Deq*, and the window constraints, instantiated for each input buffer and timestep. I^i and L^i are sequences of packet count vectors representing inputs (not enqueues) and outputs of the ith buffer over T timesteps, and B^i is the sequence of booleans representing its backlog status over time. M^{ij} is a sequence of integers representing how many packets move from input buffer i to output stream j at each timestep. We define M^i as the sequence representing the total number of packets dequeued from buffer i, where $M^i_t = \sum_{j=1}^{D} M^{ij}_t$ and D is the total number of output streams. Note that *Buff* can easily be replaced by *alternative buffer representations*, such as FPerf's, as long as that representation exposes the backlog information (B^i) and updates the buffer according to dequeue decisions (M^i).

Process captures the processing logic's state machine and its transitions over each timestep. It uses backlog information from *all* input buffers ($B = \{B^i | 1 \leq i \leq N\}$) to decide how many packets are dequeued from each ($M = \{M^i | 1 \leq i \leq N\}$). Finally, let O^j be the sequence of packet count vectors for the jth output stream. *Out* specifies the packet count vectors of each output stream ($O = \{O^j | 1 \leq j \leq D\}$) based each input buffer's output ($L = \{L^i | 1 \leq i \leq N\}$) and the state machine's decision on which input buffer sends how many packets to each output stream (M). Together, these relations define how the contention point maps input packet count vectors ($I = \{I^i | 1 \leq i \leq N\}$) to outputs ($O$).

The processing logic's state machine. We define Q as the sequence of processing logic's internal state (e.g., the most recently serviced buffer in a round-robin scheduler) over T timesteps. Let B_t and M_t be the tth elements of backlog and movement decision sequences B and M defined earlier. We encode the processing logic (*Process*) as a state machine whose the state is determined by both Q_t and B_t, extended with additional information such as buffer occupancy if needed. The relation *Init* encodes the initial states, *Trs* captures valid transitions between consecutive states, and *Act* specifies each state's associated actions determining how many packets move from each input buffer to each output stream at timestep t (M_t):

$$\begin{aligned} Process(B,M) = & \exists Q, Init(B_1, Q_1) \wedge \\ & \forall 1 \leq t < T, Trs(B_t, Q_t, B_{t+1}, Q_{t+1}) \wedge \\ & \forall 1 \leq t \leq T, Act(B_t, Q_t, M_t) \end{aligned}$$

Programming processing logic with Buffy [10]. By decoupling buffer representations from the processing logic through a well-defined interface, our modular architecture enables deriving this state machine from high-level programs written in languages such as Buffy [10]. We illustrate this using a Buffy-style program for a round-robin scheduler, which describes how data moves between N input buffers and one output stream in an abstract timestep using a constrained C-like function (Figure 6). The function interacts with buffers only through built-in primitives such as backlogged(i), which checks if the ith input buffer is non-empty, and move-p(i, j,k), which moves k packets from the ith input buffer to the jth output stream. It maintains a single global variable, turn, which records the index of the last serviced input buffer and is used to select the next non-empty one for dequeue.

To derive our state-machine encoding, we first transform the high-level program by unrolling loops, converting it into static single assignment (SSA) form [16], and expanding the built-in primitives into per-index constraints. Notably, the transformed program declares two sets of symbolic variables:

```
1 rr(int N){
2   global turn = 0;
3   local deq = false;
4   for (i in 0..N) do{
5     local next = (turn + i + 1) % N;
6     if (!deq && backlogged(next)) {
7       move-p(next, 0, 1);
8       deq = true;
9       turn = next;} } }
```

Figure 6: Buffy-style program for a Round-Robin scheduler

(1) globally-scoped variables like turn, which capture the internal state Q_t, and (2) the input buffers' backlog indicators, bl_0,...,bl_N, which correspond to B_t. Moreover, for each input buffer i, auxiliary "move" variables are introduced and initialized (move_i_j = 0). Calls to backlogged(i) are expanded into predicates over symbolic backlog variables ((i == 0 => bl_0 && ... && i == N => bl_N)), and calls to move-p(i, j, k) are expanded into assignments to the move variables.

We use this intermediate program to derive the *Act* and *Trs* relations. The relation *Act* is derived from assignments to the move variables. These assignments characterize, in terms of the symbolic variables representing Q_t and B_t, the conditions under which different numbers of packets are dequeued from each input buffer along all control-flow paths. The *Trs* relation is derived from assignments that update the processing logic's internal state (turn), together with additional consistency constraints relating backlogs across timesteps. See §B for more details, including transformed programs. We describe our experience applying this process to various schedulers in §6.3.

5 End-to-End Reasoning

Checking queries against workloads. As defined in §2.1, both workloads and queries are performance properties expressed in terms of buffer performance metrics. A performance metric pm_t is a function of buffer variables at time t. These variables include $\boldsymbol{i}_t$, $\boldsymbol{e}_t$, $\boldsymbol{o}_t$, $\boldsymbol{d}_t$, and $\boldsymbol{v}_t$, which are packet count vectors representing the packets arriving at, entering, leaving, dropped, and present in the buffer at time t, respectively (§3). Example metrics include the input count $\sigma(\boldsymbol{i}_t)$ at time t, the buffer size $\sigma(\boldsymbol{v}_t)$, and the output count $\sigma(\boldsymbol{o}_t)$.

A performance property is then defined as a set of metric sequences over time. For example, the property "the input rate for traffic class 2 should be greater than r" can be used as a workload to characterize external traffic entering a system of contention points. Intuitively, this requires that the cumulative input to the free buffers from class 2 over a fixed time period T exceeds a threshold derived from r. Formally, this corresponds to the set of input-count sequences of free buffers whose sum over time T satisfies this condition. A similar property, "throughput greater than r", defined over output-count sequences of a given buffer, can be used as a query to characterize performance questions of interest. Another example is "no drops from class 5" at a particular buffer, which corresponds to the (singleton) set of drop-count sequences in which the count for class 5 is zero at every timestep.

We check if a workload (P) applied to our contention point (CP) would imply the query (Q) by using an SMT solver like Z3 [6] to determine if the following formula is *unsatisfiable*:

$$P \wedge CP(I,O) \wedge \neg Q \tag{2}$$

If unsatisfiable, there are no inputs that satisfy the workload property while violating the query, and thus P implies Q.

Checking queries under workloads is a powerful primitive for analyzing performance properties. This is because, without restricting input traffic, performance properties rarely hold across all admissible inputs [4]. For example, one would not expect all flows to experience high throughput under arbitrary levels of cross traffic entering the network. This is in contrast to functional correctness properties, such as reachability, which are typically expected to hold independent of traffic load or intensity. A verification engine, like ours, that checks workloads against queries enables users to constrain input traffic to relevant scenarios and check if their desired performance properties hold in those settings. It can also support synthesis tools, like FPerf [4], which search for workloads that trigger performance problems by checking whether candidate workloads satisfy a given performance query.

Impact of abstraction. Our abstraction is particularly well suited for checking if a workload implies a query. To see why, note that a workload characterizes *a set* of admissible input sequences. For each such concrete input sequence, a concrete FIFO buffer produces *a single* corresponding output sequence. In contrast, our abstraction associates each input sequence with *a set* of possible output sequences. Each of the output sequences in this set satisfies the property that the difference between a packet's position in the input and output sequence is not more than the window/buffer size.

This over-approximation works well in this setting for two intuitive reasons. First, most performance metrics are aggregate statistics that are unaffected by short-term reordering within a small window. Second, workloads represent sets of input traces, not a single fixed trace. A re-ordered output sequence admitted by the abstraction for one input may correspond to the exact (or a very close) FIFO output of another input trace in the workload. Thus, the abstraction is expected to remain within the behavioral variability already captured by the workload. Consistent with this intuition, for all workloads in the case studies (§6), our abstract verification engine returned the same SAT/UNSAT result as FPerf's for Eq. 2.

Unsatisfiability. Because our abstraction is an over-approximation, if the formula in Eq. 2 is unsatisfiable (UNSAT), then the same query checked using a concrete FIFO buffer model would also return UNSAT. Intuitively, our abstract buffer allows for a superset of the behaviors of a concrete FIFO. As such, if no assignment to the free variables satisfies the formula under our model, none can satisfy it under a more precise model that admits fewer behaviors.

Satisfiability. If the formula is satisfiable (SAT), there exists an assignment to the free variables (i.e., the input traffic)

that satisfies the workload but violates the query. There are two cases here. First, the satisfying assignment found under the abstraction corresponds to a valid behavior in the precise model. In this case, checking the formula under the precise model would also return SAT. Second, the assignment is not valid in the precise model. This can occur because our abstract model is an over-approximation and allows for more behaviors. In this case, checking the formula with the precise model may return SAT or UNSAT – both are possible.

Consider a rate limiter that dequeues one packet per timestep from its single input buffer. The buffer and window size are four, and both are full with packets of class red r at $t = 1$. For $t \geq 1$, one packet arrives in each timestep, alternating between traffic classes blue (b) and pink (p). Suppose the property states that for $t > 4$ (i.e., after all r packets are drained), the output stream contains at least one b packet every two timesteps. This property trivially holds under a precise buffer model (UNSAT). However, our abstract model returns SAT and produces a counterexample. This is because after four timesteps, the buffer contains two b and two p packets. The window constraints ensure two b and two p packets in timesteps 5 to 8, but do not constrain their order. So, the output sequence r, r, r, r, b, p, p, b is permitted by the abstraction and returned as a counter example for property violation, even though it cannot occur in the precise buffer model.

The particular mismatch happens because the granularity of the property (2 packets) is less than the window size (4 packets). In our empirical case studies, we did not encounter such cases, and in general, any other spurious SAT results. As discussed earlier, we attribute this to our abstraction aligning well with workloads and properties being checked.

6 Evaluation

We demonstrate how our approach improves verification time and highlight the value of its modular architecture in enabling interchangeable buffer abstractions and systematic derivation of contention point state-machines. First, we apply our abstraction to FPerf's case studies (Table 1) and show how our abstraction significantly improves verification time for realistic buffer sizes while producing consistent answers to FPerf's workload verification questions. In fact, verification time under our abstraction remains largely insensitive to buffer size. This is particularly important as tools like FPerf can make 100s of calls to the verification engine to synthesize workloads. So, larger verification times would significantly degrade their usability. Second, we write Buffy programs for the contention points in the case studies and use the approach described in §4 to manually derive the state machines for our model. We leave automated generation of SMT formulas from Buffy programs to future work.

To compare with FPerf, we run FPerf's search algorithm for each case study across a range of buffer capacities and record the workloads for which FPerf invokes its verification engine. We then verify the same workloads using our model

	Prio	RR	FQ-CoDel	Loom
Buffers	4	5	5	14
Boolean Vars	56	100	350	280
Integer Vars	336	600	980	5210
Timesteps	7	10	14	10

Table 1: Case study statistics for our abstract model. FPerf can use up to 17K variables for the same case studies.

and check whether the results (SAT/UNSAT) are consistent with FPerf. We also compare the average and 95th-percentile time required to verify workloads in FPerf and our model. We use the recommended window size equal to buffer size for our abstraction, and where applicable, also report verification time without window constraints. Similar to FPerf, we use Z3 as our backend. All experiments are performed on a server with an Intel Xeon Gold 6544Y 3.6GHz CPU with 32 cores.

The implementation of our verification tool, *Count-Buffy*, and the case studies are available on GitHub [17].

6.1 Packet schedulers

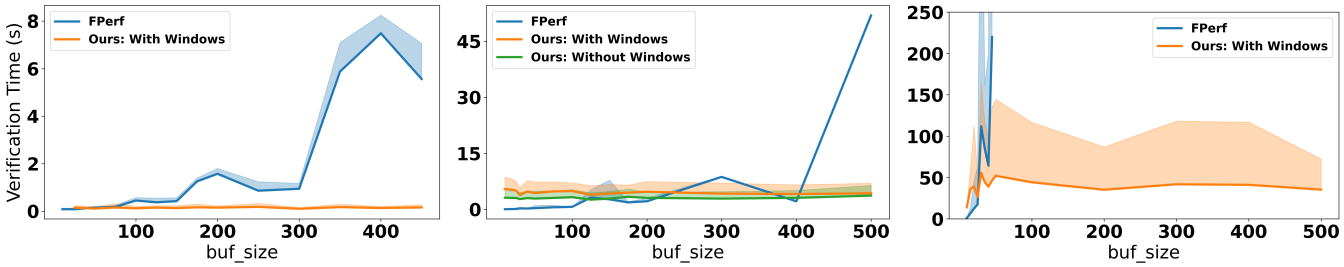
For all the following case studies, our verification results (SAT/UNSAT) were consistent with FPerf for all workloads.

Strict-Priority. This scheduler has four input buffers $i_1, \cdots, i_4$, with i_1 having the highest and i_4 the lowest priority. At each timestep, the scheduler dequeues one packet from the highest-priority backlogged buffer. The query asks if i_3 can remain blocked for more than five timesteps, where blocked means the buffer is backlogged but not selected to dequeue packets. Figure 1 compares the average and tail verification times of FPerf and our model across workloads generated during FPerf's search, for buffer sizes from 10 to 500 packets (the shaded region shows the gap between average and tail). Starting at buffer size of 100 packets, FPerf's verification time significantly increases, while our abstract model's verification time remains stable, answering queries in under 200ms.

In this case study, packets in different buffers may belong to different traffic classes, but packets within each buffer share the same priority class. As such, $K = 1$ for our abstraction, and we can forgo the window constraints. The results without window constraints follow a similar trend and are not shown, as they are visually indistinguishable from those with windows, given FPerf's high verification times.

Round-Robin. This scheduler has 5 input buffers $i_1, \cdots, i_5$ and one output stream. At each timestep, it dequeues one packet from the next backlogged buffer in round-robin order. Following FPerf, we constrain the inputs using a "base workload" that requires each input buffer to receive at least 4 packets every 5 timesteps. We analyze the scheduler for 10 timesteps, and check if the difference between dequeues from i_2 and i_3 exceeds 2, indicating a fairness problem.

Figure 7a compares verification times between FPerf and our abstract model for buffer sizes between 10 and 450 packets for FPerf's candidate workloads. The results are consistent with the previous case study – starting at buffer size 75, the



(a) Round-Robin Scheduler (b) FQ-CoDel Scheduler (c) Compositional Case-Study

Figure 7: Verification time comparison between our approach and FPerf for increasing buffer capacities (shaded = p95)

gap between the two models grows significantly. FPerf's verification time increases to $\sim 8\,s$. while ours remains consistently under $350\,ms$ even for buffers of 450 packets. Similar to the priority scheduler, we can forgo window constraints, and we observe similar trends without them.

FQ-CoDel. This scheduler is inspired by a simplified version of the scheduler in FQ-CoDel [4, 13]. At a high level, it services buffers in a mostly round-robin fashion, but prioritizes the first few packets of newly backlogged buffers to reduce latency for short flows. The scheduler keeps two lists of buffer pointers: newly backlogged (*new_bufs*) and persistently backlogged buffers (*old_bufs*). On each dequeue, it selects the buffer at the head of *new_bufs*, say i_h, to send a packet. If i_h becomes empty, it is removed from *new_bufs* and marked inactive. If it remains non-empty but is no longer considered newly backlogged, it is moved to *old_bufs*. Otherwise, it is placed at the end of *new_bufs*. When *new_bufs* is empty, the scheduler instead serves the buffer at the head of *old_bufs*.

This case study models this scheduler with five buffers $i_1, \cdots, i_5$ and checks for a subtle issue that the RFC [13] warns about: since we immediately deactivate a buffer and remove it from *new_bufs* when it becomes empty, it can reenter *new_bufs* as soon as it receives a new packet. By sending at just the right rate, such a flow can repeatedly regain priority and starve the flows in *old_bufs*. To check for this issue, we use a base workload that requires the first four buffers to be persistently backlogged in the first 14 timesteps. The query asks whether i_5 can get more than its fair share of service by checking if its number of dequeues can exceed $2 \times \lfloor \frac{T}{5} \rfloor$.

Figure 7b compares verification times between FPerf and our abstract model with and without window constraints for buffer sizes between 10 to 500 packets for FPerf's candidate workloads. The three buffer encodings perform similarly up to buffer sizes of 400 packets, where FPerf's verification time has a significant jump to over $45\,s$, while both our abstract encodings remain consistently below $5.5\,s$. The gap between the encodings with and without constraints is more pronounced in this case study: without window constraints, verification is on average $1.3\,s$ faster across most buffer sizes.

Very large buffers. To evaluate the scalability of our abstraction beyond buffer sizes of 500 packets, we verified this case study's workloads for buffer sizes up to 10^9 packets and observed no meaningful impact on verification time.

Takeaways. Our abstract buffer encoding results in consistently low verification times independent of buffer capacities, whereas FPerf's verification time, even for stand-alone contention points, varies significantly with increasing buffer capacities. Empirically, we did not observe inconsistencies between FPerf's verification answers, based on a precise buffer model, and ours, based on an abstract one (see §5). Finally, different buffer models are effective in different settings. For example, for smaller buffers (10s of packets), a fully "spelled-out" buffer representation may perform better due to the overheads of integer arithmetic in count-based models. For larger buffers, our abstraction is more efficient. When packets from different traffic classes do not mix in buffers (e.g., stand-alone packet schedulers), we can further simplify the abstract model by using counts but without windows. These results highlight the value of a modular contention point model that enables switching between buffer representations as needed.

6.2 Composition of Contention Points

Loom. Inspired by Loom [18], this case study analyzes a composition of contention points (Figure 8). Suppose two tenants (T1 and T2) share four CPU cores on a machine. Both run the Spark application [19], while T2 also runs Memcached [20]. A round-robin scheduler runs on each core to share network resources evenly between T1 and T2. The NIC scheduler merges packets from the per-core TX queues into one stream, which is then sent to a scheduler that prioritizes Memcached traffic over Spark. Our buffers track three traffic classes: T1 Spark, T2 Spark, and T2 Memcached. Similar to FPerf, we include classifiers between contention points that (de)multiplex output traffic streams based on class and forward them to the appropriate input buffers of downstream contention points.

Our scheduler has five buffers $i_1, \cdots, i_5$. We use a base workload that ensures each tenant's Spark application receives a steady stream of inputs across all four cores. The query checks if T2 can send at least three packets more than T1 within 10 timesteps, indicating a fairness violation. Figure 7c compares verification times between our model and FPerf for FPerf's candidate workloads across increasing buffer sizes. FPerf's verification time grows substantially for realistic buffer sizes. For a buffer size of 30, FPerf's average verification time exceeds $200\,s$ while ours stays consistently around $50\,s$ even for buffer sizes up to 500.

Small leaf-Spine. Our abstract buffer encoding scales well with buffer size. But, it is sensitive to the number of traffic classes as buffer constraints operate over vectors of packet counts whose size equals the number of classes. So, increasing the number of classes increases encoding size. In contrast, FPerf is sensitive to buffer size and not number of traffic classes. It represents each buffer slot using a boolean occupancy variable and metadata fields that travel with packets but are not directly manipulated in buffer operations. As such, increasing the number of traffic classes mainly expands metadata ranges and has little impact on encoding size.

To explore this difference in trade-offs, we use a small leaf-spine topology with six hosts connected via two spine and three leaf input-queued switches as contention points, each with virtual output queues (VOQs) and an iSlip crossbar scheduler [14]. Each packet is associated with metadata fields *dst* (destination host) and *ecmp* (spine choice). In our model, each combination forms a traffic class, resulting in $K = 12$ classes for all the buffers in the network. Our query checks if the throughput between hosts 1 and 6 can drop below line-rate using packet counts at the buffer leading to host 6 over 10 timesteps. The base workload requires host 1 to send a steady stream of traffic to host 6, while no two hosts send traffic to the same destination (permutation traffic matrix).

Without optimizations, for buffer size 10, workload verification takes over 30 minutes in our model, compared to $\sim 1,s$ in FPerf. While FPerf is expected to perform better for small buffers (§6.1), this gap is excessive. However, routing-aware optimizations can reduce the number of traffic classes and achieve performance comparable to FPerf, even for small buffers. Using workload constraints on *dst* and *ecmp* together with leaf–spine routing rules, we eliminate, for each buffer, traffic classes that can never reach it. For example, if constraints imply $dst = 5$ for host 3's traffic, the VOQ buffer for traffic from host 3 to leaf 2 can contain only two classes ($dst = 5$, $ecmp \in \{1,2\}$). Similarly, traffic from spine 1 to leaf 1 must have $dst \in \{1,2\}$, so each VOQ from spine 1 to a host attached to leaf 1 carries only one class. This reduces the number of traffic classes to an average of 1.09 per buffer, lowering the verification time to $\sim 1.3\,s$ on average, comparable to FPerf for a small buffer size of 10. We expect similar routing-aware optimizations to apply in other settings.

6.3 From Buffy programs to state machines

Our modular contention-point architecture (1) supports interchangeable buffer encodings (benefits demonstrated in §6.1), and (2) enables systematic derivation of the processing logic state machine from programs written in high-level languages such as Buffy [10]. To confirm the second benefit, we wrote Buffy programs for the schedulers in §6.1 and §6.2 and derived their state machines using the transformation process in §4. In all cases, the process involved mostly straightforward transformations and no scheduler-specific modifications. The programs and their transformations are provided in §B.

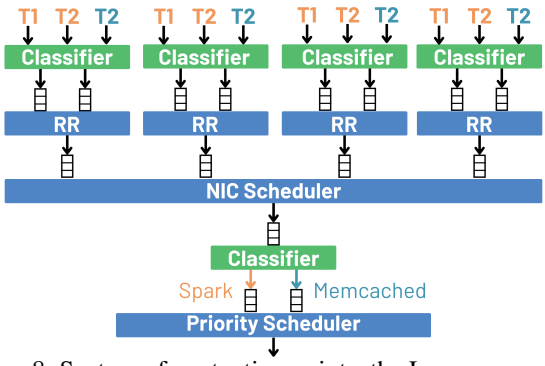


Figure 8: System of contention points, the Loom case-study

Priority and Round-Robin. We described the transformation for a round-robin scheduler with two input buffers in §4. For the round-robin scheduler in §6.1, we followed the same process with five input buffers. Unlike round-robin, the priority scheduler's processing logic maintains no internal state – its decisions depend only on input-buffer backlogs. Accordingly, the derived state-machine relations depend only on B_t (there is no Q_t). Moreover, after loop unrolling, the arguments to backlogged and move become constants. Thus, unlike round-robin, where the arguments depend on symbolic variables, no expansion was needed for these primitives.

FQ-CoDel. This program uses Buffy's built-in list construct to track newly and persistently backlogged buffers with two lists of size N, where N is the number of input buffers (§6.1). Deriving the state-machine encoding therefore also requires translating list operations. Each list lst is replaced with symbolic variables lst_0,...,lst_N. These variables encode the relative order of buffers in the list: a larger value means the buffer was inserted earlier and is closer to the front. List operations become predicates or assignments over these variables. A call to lst.enq(i) is transformed to assignments that increment lst_i and all positive lst_j to preserve ordering. A call to lst.has(i) becomes lst_i > 0. Finally, nq.pop_front() expands into checks and assignments that find the buffer with the largest lst_i, store the buffer index in an auxiliary variable, and reset lst_i to zero. If any arguments are symbolic, they are expanded further, similar to backlogged and move.

Loom. The contention points in Loom (§6.2) are variations of round robin and priority schedulers, and their state machines are derived using the same process.

Translation remarks. Our high-level syntax differs slightly from Buffy (e.g., move-p takes buffer indices rather than buffer objects; see §B). These differences did not affect the schedulers we evaluate, which rely only on Buffy's core buffer primitives. Thus, we could systematically translate Buffy programs into our syntax and derive the corresponding state-machine encodings (§4). We performed this translation manually and leave automation to future work.

7 Discussion

Reasoning about delay. Our count-based abstraction does not directly support reasoning about per-packet metrics. For

example, it cannot track the delay experienced by individual packets or answer precise per-packet delay queries. Instead, delay properties can be expressed in terms of buffer backlogs. For instance, to determine whether end-to-end delay may exceed a threshold X, one can ask whether the sum of buffer backlogs exceeds a corresponding threshold derived from X.

Active Queue Management (AQM). Our approach can be extended to support AQM and per-hop flow control. Conceptually, these algorithms monitor congestion-related metrics, such as buffer occupancy, and react when these metrics exceed predefined thresholds by marking or dropping packets (e.g., RED [21] and ECN [22]) or signaling upstream senders (e.g., PFC [23]). For many such mechanisms, the primary signal is buffer occupancy, which is modeled precisely in our framework. Reactions can be incorporated by extending the buffer abstraction or within the processing logic state machine.

For example, to model packet marking, each traffic class can be encoded as two subclasses, marked and unmarked. The enqueue logic can be extended to non-deterministically change the class of a subset of incoming packets from unmarked to marked, with the subset size reflecting the desired marking probability. Packet drops on enqueue can be modeled similarly. Per-hop flow control mechanism such as PFC can be supported by (1) extending the interface between buffers and the processing logic to include not only whether a buffer is backlogged, but also if its occupancy has passed a predefined threshold, (2) extending the processing logic action to allow adding new control packets (e.g., pause and resume) to output streams in addition to moving packets there from input buffers. Upstream behavior can be modeled using existing primitives by maintaining internal state that tracks paused queues and updating this state upon receiving control packets. AQM schemes that rely on per-packet metrics to detect persistent queue buildup (e.g., CoDel [24]) cannot be directly captured in our abstraction, as discussed above.

Priority queues. Priority queues, as opposed to FIFO buffers, are not directly supported by our abstraction as they require tracking per-packet ordering. For example, a PIFO [25] assigns a rank to each packet and dequeues packets in rank order (similar to virtual-clock-based schedulers). However, such priority queues are uncommon in practice, especially in high-speed network devices, which typically rely on combinations of priority and round-robin schedulers. Moreover, prior work such as SP-PIFO [26] shows that PIFOs can be approximated using multiple FIFOs and a strict-priority scheduler.

Application to other domains. While this paper focuses on contention points in packet processing, the underlying abstractions of buffers and mediation logic arise in many other domains, including task scheduling in operating systems [27], cluster scheduling [28], and storage and I/O scheduling [29]. We therefore envision our approach as broadly applicable for analyzing contention-related properties in settings where resources are managed through queues and mediation policies.

8 Related Work

We have discussed FPerf [4] and Buffy [10] throughout the paper. Here, we focus on other related work for analytical and symbolic reasoning about network performance.

Logic-based approaches. Quasi [30] answers queue-related queries in the context of given coarse-grained measurements of per-port packet counts on a network switch. It uses a layered approach, where a fast conservative static analysis is followed (if needed) by a precise SMT-based procedure. Although it too uses count-based abstractions for queues, it does not consider different traffic classes. It focuses mostly on checking *consistency* with per-port packet count measurements, rather than on reasoning about expressive properties such as fairness. Furthermore, it does not model (or provide support for interfacing with) the decision logic in contention points, or for composing them in a system. CCAC [5] models Internet paths as a generalized non-deterministic token bucket filter (TBF), and encodes in SMT to reason about a flow's behavior under a given congestion control algorithm. While CCAC also uses integers to represent the number of bytes in the TBF's buffer, it is focused solely on analyzing congestion control algorithms and not contention points in general.

Network Calculus. Network calculus [1, 2] introduces a rigorous mathematical framework for analyzing performance guarantees for a flow over a sequence of network devices. However, to use the calculus, the functionality of network devices and the input workload should be approximated by well-defined mathematical functions that can be used in $(min, +)$ algebra. These functions must be reasonably concise and have tight approximation bounds for the final performance guarantee to be realistic and useful. Deriving such functions is challenging, particularly for today's complex network functionality and traffic patterns [3].

9 Conclusion

Formal methods can unlock new means for analyzing the performance of network contention points, provided they can be made to perform well enough on a range of practical systems. In this work, we introduced flexible new abstractions for modeling buffers, the central data structure in any contention point. We also explain how to craft interfaces between buffer abstractions and contention point decision logic, to enable engineers to try out different abstractions. We evaluate the performance of our abstractions and demonstrate that they scale independently of buffer size, improving the performance of analyses on a range of practical examples with larger buffers by an order of magnitude or more relative to the state-of-the-art.

Acknowledgments

We thank our shepherd, Peng Zhang, and the anonymous reviewers for their helpful feedback. This work was supported in part by a Canada Research Chair grant CRC-2023-00035, an NSERC Discovery grant RGPIN-2023-03775, and NSF grants CNS-2319442 and CNS-2312539.

References

[1] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2001.

[2] Rene L Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Transactions on information theory*, 1991.

[3] Florin Ciucu and Jens Schmitt. Perspectives on network calculus: no free lunch, but still good value. In *ACM SIGCOMM*, 2012.

[4] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. Formal methods for network performance analysis. In *USENIX NSDI*, 2023.

[5] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *ACM SIGCOMM*, 2021.

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[7] Cisco Nexus 9300 Platform Switches. Data sheet. https://www.cisco.com/. Accessed: March 2026.

[8] Cisco Nexus 9800 Series Switches. Data sheet. https://www.cisco.com/. Accessed: March 2026.

[9] Eric Dumazet and Vijay Subramanian. *tc-fq-codel(8) — Linux manual page*. Linux Kernel. Accessed: March 2026.

[10] Amir Seyhani, Junyi Zhao, Aarti Gupta, David Walker, and Mina Tahmasbi Arashloo. Buffy: A Formal Language-Based Framework for Network Performance Analysis. In *ACM HotNets*, 2024.

[11] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[12] The Linux Documentation Project. Chapter 2. classless queueing disciplines. https://tldp.org/HOWTO/Traffic-Control-HOWTO/classless-qdiscs.html. Accessed: March 2026.

[13] Toke Hoeiland-Joergensen, Paul McKenney, Dave Taht, Jim Gettys, and Eric Dumazet. The flow queue codel packet scheduler and active queue management algorithm. Internet-draft, Internet Engineering Task Force, 2018.

[14] Nick McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 2002.

[15] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *ACM SIGCOMM*, 1995.

[16] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1991.

[17] Count-Buffy GitHub repository. https://github.com/all-things-networking/count-buffy, 2026. Accessed: 2026-03.

[18] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient NIC packet scheduling. In *USENIX NSDI*, 2019.

[19] Apache Spark. https://spark.apache.org/. Accessed: March 2026.

[20] Memcached: a Distributed Memory Object Caching System. http://www.memcached.org/. Accessed: March 2026.

[21] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 2002.

[22] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, IETF, 2001.

[23] IEEE Standards Association. Priority based flow control. Technical Report 802.11Qbb., 2011.

[24] Kathleen Nichols, Van Jacobson, Andrew McGregor, and Jana Iyengar. Controlled Delay Active Queue Management. RFC 8289, IETF, 2018.

[25] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *ACM SIGCOMM*, 2016.

[26] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out behaviors using Strict-Priority queues. In *USENIX NSDI*, 2020.

[27] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts*. John Wiley & Sons, 2019.

[28] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *ACM EuroSys*, 2015.

[29] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for μs latency and high throughput. In *USENIX OSDI*, 2021.

[30] Divya Raghunathan, Maria Apostolaki, and Aarti Gupta. A Layered Formal Methods Approach to Answering Queue-related Queries. In *USENIX NSDI*, 2025.

A Vector operations

We define the following vector operations for vectors $\boldsymbol{v}$ and $\boldsymbol{v}'$:

$$\begin{aligned}
\boldsymbol{v} \leq \boldsymbol{v}' &\iff \forall 1 \leq i \leq K, v_i \leq v'_i \\
\boldsymbol{v} + \boldsymbol{v}' &= \langle v_1 + v'_1, \ldots, v_K + v'_K \rangle \\
\boldsymbol{v} = \boldsymbol{v}' &\iff \forall 1 \leq i \leq K, v_i = v'_i \\
\boldsymbol{v} \leq \boldsymbol{v}' &\iff \forall 1 \leq i \leq K, v_i \leq v'_i \\
\sigma(\boldsymbol{v}) &\iff \sum_{1 \leq i \leq K} v_i \\
\boldsymbol{v} = n &\iff \sigma(\boldsymbol{v}) = n
\end{aligned}$$

B Programming processing logic with Buffy

In this section, we present Buffy-style programs for the case studies in §6 and the transformed intermediate programs used to derive the *Trs* and *Act* relations of their corresponding processing logic state machines (§4, §6.3).

Buffy programs. A Buffy program describes how data moves between buffers in a time step. Here, a time step is a single execution of the program – the granularity of time can change depending on what is being modeled. For a scheduler, for example, it can be the time it takes to do an enqueue or a dequeue operation. The program takes in one or more buffers as input and one or more write-only buffers as output. Users specify the logic for moving data from the input buffers to the output buffers using a constrained, imperative C-like language.

In Buffy programs, there are no pointers, loops have concrete bounds, and the only interaction with buffers is through the buffer interface. This interface allows users to get the size of the backlog of input buffers in packets or bytes (backlog-p and backlog-b) and move a specific number of packets or bytes from an input buffer to an output buffer (move-p and move-b). The output buffers are "flushed" after each program execution (into the input buffer of another program in case of composition). As such, they are empty at the start of the next execution. Variables declared as global will persist across executions. The scope of the others is limited to individual executions of the program.

Our Buffy-style programs. Our high-level programs differ slightly from Buffy:

- In Buffy, move-* takes abstract buffer objects as parameters, whereas our programs use buffer indices.
- In Buffy, backlog-*(b) returns the backlog of a buffer object b whereas our programs use backlogged(i) to return a boolean indicating whether the input buffer with index i is non-empty.
- Our programs assume output streams rather than output buffers. In Buffy, outputs are write-only buffers that are flushed after each execution and do not retain packets across timesteps. Thus, they are equivalent to streams in our model.

The schedulers evaluated in this paper do not require the full set of Buffy primitives, such as actual backlog size or buffer filtering. As a result, translating their Buffy programs into our high-level syntax was straightforward. As such, transforming their Buffy programs to our high-level syntax was straightforward. Buffy-style programs with our high-level syntax were then transformed to the intermediate programs described in §4 and systematically mapped to the state-machine encoding.

Outline. The Buffy-style programs and their transformations for priority, round-robin, and the simplified FQ-CoDel scheduler are described in §B.1, §B.2, and §B.1. The contention points in Loom (§6.2) are variations of the round robin and priority schedulers, and their state machines can be derived similarly.

B.1 Priority scheduler program transformation

Listing 1 shows the Buffy-style program for our priority scheduler, and Listing 2 is its transformed version for $N = 2$ input buffers. Since this scheduler has no global state, any transition between states is possible, so the Trs relation only depends on B_t and include consistency constraints relating backlogs across timesteps. One such constraint for a scheduler that only dequeues one packet per timestep is that at most one buffer can go from being backlogged to not backlogged from one state to another. The Act relation is derived from the assignments to the *move* variables. Lines 26 and 27 show the assignments to the last version of the move variables in the unrolled program and can be used to derive the Act relation. Expressions in the right-hand side of these assignments can be unfolded until they are expressed in terms of backlog and global variables, which correspond to B_t and Q_t vectors in the Act relation.

```
1 prio(int N){
2   local dequeued = false;
3   for (i in 0..N) do{
4     if (!dequeued && backlogged(i)) {
5       move-p(i, 0, 1);
6       dequeued = true;
7     }
8   }
9 }
```

Listing 1: Buffy-style program for the priority schedule

```
1 // symbolic variables
2 bool backlogged_0, backlogged_1;
3 // Boilerplate to initialize helper move vars
4 int move0_0_0 = 0;
5 int move0_1_0 = 0;
6 bool deq0 = false;
7 // --- Iteration 0
8 // Conditional -- condition
9 bool cond0 = !deq0 && backlogged_0;
10 // Conditional -- body
11 int move1_0_0 = 1;
12 int move1_1_0 = 0;
13 bool deq1 = true;
14 // Conditional -- after
15 int move2_0_0 = cond0 ? move1_0_0 : move0_0_0;
16 int move2_1_0 = cond0 ? move1_1_0 : move0_1_0;
17 bool deq2 = cond0 ? deq1 : deq0;
18 // Iteration 1
19 // Conditional -- condition
20 bool cond1 = !deq2 && backlogged_1;
21 // Conditional -- body
22 int move3_0_0 = move2_0_0; // Buffer 0 move count stays what it was
23 int move3_1_0 = 1;
24 bool deq3 = true;
25 // Conditional -- after
26 int move4_0_0 = cond1 ? move3_0_0 : move2_0_0;
27 int move4_1_0 = cond1 ? move3_1_0 : move2_1_0;
28 bool deq4 = cond1 ? deq3 : deq2;
```

Listing 2: Transformed Buffy program for the priority scheduler and $N = 2$

B.2 Round-Robin scheduler program transformation

Figure 6 shows the Buffy-style program for the round-robin scheduler and Listing 3 shows the transformed version for $N = 2$. The transformation of the round-robin scheduler is described in §4.

```
1 // symbolic variables
2 int turn0;
3 bool backlogged_0, backlogged_1;
4 // Boilerplate to initialize helper move vars
5 int move0_0_0 = 0;
6 int move0_1_0 = 0;
7 bool deq0 = false;
8 // Iteration 0
9 int next0 = (turn0 + 1) % 2;
```

```
10 // Conditional -- condition
11 bool bl_next0 = (next0 == 0 => backlogged_0) &&
12                 (next0 == 1 ==> backlogged_1);
13 bool cond0 = !deq0 && bl_next0;
14 // Conditioanl -- body
15 int move1_0_0 = next0 == 0 ? 1 : 0;
16 int move1_1_0 = next0 == 1 ? 1 : 0;
17 bool deq1 = true;
18 int turn1 = next0;
19 // Conditional -- after
20 int move2_0_0 = cond0 ? move1_0_0 : move0_0_0;
21 int move2_1_0 = cond0 ? move1_1_0 : move0_1_0;
22 bool deq2 = cond0 ? deq1 : deq0;
23 int turn2 = cond0 ? turn1: turn0;
24 // Iteration 1
25 int next1 = (turn0 + 2) % 2;
26 // Conditional -- condition
27 bool bl_next1 = (next1 == 0 ==> backlogged_0) &&
28                 (next1 == 1 ==> backlogged_1);
29 bool cond1 = !deq2 && bl_next1;
30 // Conditional -- body
31 int move3_0_0 = next1 == 0 ?  1 : move2_0_0;
32 int move3_1_0 = next1 == 1 ?  1 : move2_1_0;
33 bool deq3 = true;
34 int turn3 = next1;
35 // Conditional -- after
36 int move4_0_0 = cond1 ? move3_0_0 : move2_0_0;
37 int move4_1_0 = cond1 ? move3_1_0 : move2_1_0;
38 bool deq4 = cond1 ? deq3 : deq2;
39 int turn4 = cond1 ? turn3 : turn2;
```

Listing 3: Transformed program for the round-robin scheduler for $N = 2$

B.3 FQ-CoDel scheduler program transformation

Listing 4 shows the Buffy-style program for the FQ-CoDel scheduler. The program uses Buffy's built-in list construct to track new and old buffers (see §6.1) with *nq* and *oq*, respectively. As described in §6.3, each list is represented by a set of variables, one for tracking the index of each input buffer within the list. The transformation relies on the assumption that at most one pointer to each buffer can be present in a list at any point in time, which is true for this scheduler. The variables for the two lists correspond to the scheduler's internal state Q_t. Listing 5 shows the transformed program for $N = 2$.

```
1  global list nq;
2  global list oq;
3  fq(int N){
4    for (i in 0..N) do{
5      if (backlogged(i) & !oq.has(i) & !nq.has(i)){
6        nq.enq(i);
7      }
8    }
9    local head = -1;
10   if(!nq.empty()){
11     head = nq.pop_front();
12     move-p(head, 0, 1);
13   }
14   else if (!oq.empty()) {
15     head = oq.pop_front();
16     move-p(head, 0, 1);
17   }
18   if (head != -1 && backlog-p(head)){
19     oq.push_back(head);
20   }
21 }
```

Listing 4: Buffy-style program for FQ-CoDel scheduler

```
1 // Symbolic variables
2 int nq0_0, nq0_1 = -1, -1;
3 int oq0_0, oq0_1 = -1, -1;
4 bool backlogged_0, backlogged_1;
```

```
5  // Boilerplate to initialize helper move vars
6  int move0_0_0 = 0;
7  int move0_1_0 = 0;
8
9  // Loop start
10 // -- Loop Iteration 0
11 // Conditional -- condition
12 bool cond0 = backlogged_0 && ! (oq0_0 >= 0) && ! (nq0_0 >= 0);
13 // Conditional -- body
14 int nq1_0 = nq0_1 >= 0 ? 1 : 0;
15 int nq1_1 = nq0_1;
16 // Conditional -- after
17 int nq2_0 = cond0 ? nq1_0 : nq0_0;
18 int nq2_1 = cond0 ? nq1_1 : nq0_1;
19 // --- Loop Iteration 1
20 // Conditional -- condition
21 bool cond1 = backlogged_1 && ! (oq0_1 >= 0) && ! (nq2_1 >= 0);
22 // Conditional -- body
23 int nq3_0 = nq2_0;
24 int nq3_1 = nq2_0 >= 0 ? 1 : 0;
25 // Conditional -- after
26 int nq4_0 = cond1 ? nq3_0 : nq2_0;
27 int nq4_1 = cond1 ? nq3_1 : nq2_1;
28 // Loop end
29
30 int head0 = -1;
31
32 // Conditional (if) -- condition
33 bool cond3 = !(nq4_0 < 0 && nq4_1 < 0);
34 // Conditional (if) -- body
35 int head1 = nq4_0 == 0 ? 0 : 1;
36 int nq5_0 = nq4_0 - 1;
37 int nq5_1 = nq4_1 - 1;
38 int move1_0_0 = (head1 == 0) ? 1 : move0_0_0;
39 int move1_1_0 = (head1 == 1) ? 1 : move0_1_0;
40
41 // Conditional (else) -- condition
42 bool cond4 = !(oq0_0 < 0 && oq0_1 < 0);
43 // Conditional (else) -- body
44 int head2 = oq0_0 == 0 ? 0 : 1;
45 int oq1_0 = oq0_0 - 1;
46 int oq1_1 = oq0_1 - 1;
47 int move2_0_0 = (head2 == 0) ? 1 : move0_0_0;
48 int move2_1_0 = (head2 == 1) ? 1 : move0_1_0;
49
50 // Conditional  -- after
51 int head3 = cond3 ? head1 : (cond4 ? head2 : head0);
52 int nq6_0 = cond3 ? nq5_0 : nq4_0;
53 int nq6_1 = cond3 ? nq5_1 : nq4_1;
54 int oq2_0 = (!cond3 && cond4) ? oq1_0 : oq0_0;
55 int oq2_1 = (!cond3 && cond4) ? oq1_1 : oq0_1;
56 int move3_0_0 = cond3 ? move1_0_0 : (cond4 ? move2_0_0 : move0_0_0);
57 int move3_1_0 = cond3 ? move1_1_0 : (cond4 ? move2_1_0 : move0_1_0);
58
59 // Conditional -- condition
60 bool cond5 = (head3 != -1) && (head3 == 0 => backlogged_0) && (head3 == 1 => backlogged_1) ;
61 // Conditional -- body
62 int oq3_0 = head3 == 0 ? ( oq2_1 >= 0 ? 1 : 0) : oq2_0;
63 int oq3_1 = head3 == 1 ? ( oq2_0 >= 0 ? 1 : 0) : oq2_1;
64 // Conditional -- after
65 int oq4_0 = cond5 ? oq3_0 : oq2_0;
66 int oq4_1 = cond5 ? oq3_1 : oq2_1;
```

Listing 5: Transformed program for FQ-CoDel scheduler for $N = 2$