Manuscript Under Submission

Automated Network Performance Analysis with Workload Synthesis

Mina Tahmasbi Arashloo
Cornell University

Ryan Beckett
Microsoft Research

Rachit Agarwal
Cornell University

Abstract

Accurate and thorough analysis of network performance is challenging. Network simulations and emulations can only cover a subset of the growing set of workloads networks can experience, leaving room for unexplored corner cases and bugs that can cause sub-optimal performance on live traffic. Techniques from queuing theory and network calculus can provide rigorous bounds on performance metrics, but typically require the behavior of network components and the arrival pattern of traffic to be approximated with concise and well-behaved mathematical functions. As such, they are not immediately applicable to emerging workloads and the new algorithms and protocols developed for handling them.

In this paper, we propose to use formal methods to automatically reason about network performance. We show that it is possible to accurately model network components and their queues in logic, and use techniques from program synthesis to automatically generate concise interpretable workloads as answers to queries about performance metrics. As such, our approach offers a new point in the space of existing tools for analyzing network performance: it is more exhaustive than simulation and emulation, and can be readily applied to algorithms and protocols that are expressible in first-order logic. We demonstrate the effectiveness of our approach by analyzing scheduling algorithms and generating concise workloads that can cause throughput, fairness, and starvation problems.

1 Introduction

Modern networks serve a diverse set of applications, each with different traffic patterns and network performance requirements in terms of throughput, latency, jitter, and loss [3,17,42,45]. As such, operators constantly adapt network protocols and algorithms, e.g., for packet scheduling, congestion control, load balancing, and routing, to match the emerging application needs. These protocols and algorithms are often analyzed, both empirically and analytically, to predict how they would perform on different kinds of input workloads and traffic patterns. However, it is not uncommon for operators to miss corner cases or bugs that can lead to sub-optimal or poor network performance for some subsets of traffic when these protocols and algorithms are in production and processing live traffic [10,16,29,43,56].

To see why, consider the existing empirical approaches to network performance analysis. Operators can use simulators, emulators, and testbeds [4,5,40] to experiment with a workload over a faithful model of network components and their corresponding algorithms and protocols at the packet level and with great accuracy. However, they cannot possibly experiment with all kinds of workloads that a network may face in production and have to pick and choose which ones to try. This leaves room for unexplored workloads and traffic patterns that may experience poor performance because of overlooked corner cases and bugs. Techniques from queuing theory and network calculus can provide rigorous bounds on performance metrics [8,19,27,36,41,51], but typically require the behavior of network components and the arrival pattern of traffic to be approximated with concise and well-behaved mathematical functions. As such, they are not immediately applicable to emerging workloads and the new algorithms and protocols developed for handling them [14].

We propose an alternative approach, to use formal methods for analyzing network performance. Our main insight is to use *syntax-guided synthesis* (SyGuS) [21,26,44,47,49,54] to automatically generate *workloads* in response to queries about network performance.That is, operators can specify the network components they are interested in, the protocols and algorithms running in those components, and performance metrics of interest such as throughput, latency, jitter, or loss. They can then ask what kinds of input traffic patterns, or workloads, will cause performance metrics to drop below or rise above certain thresholds.

To answer the query, we first use logical variables and constraints to encode packets, how they are processed in network components, and how performance metrics change over time, all in satisfiability modulo theories (SMT). We define a language for specifying workloads as sets of packet traces. We then systematically search through the space of all possible workloads expressible in that language to find one where *all* packet traces represented by that workload can satisfy the query and experience sub-optimal or poor performance. If yes, that workload is returned as the answer to the query.

Why synthesize workloads? Outputting a single packet sequence that causes sub-optimal performance is not always useful as it specifies the ordering and timing of every single packet and is not easily interpretable. Instead, workloads identify entire classes of traffic that can cause performance problems and present them to operators in concise meaningful representations. Moreover, workload synthesis nicely complements existing empirical and theoretical approaches for performance analysis. Unlike empirical analysis, there is no need to try every single packet trace separately to see whether it can result in performance problems. Unlike existing theoretical approaches, there is no need to grapple with approximating network behavior with concise and well-behaved, yet representative functions; the details can all be faithfully modeled and reasoned about in logic.

Making workload synthesis tractable is quite challenging. First, performance metrics (e.g., throughput and latency) are statistics over packet streams and are affected by the order and the time at which packets from competing traffic enter and exit network components. We need to reason about such interactions efficiently, so that we can check, in reasonable time, whether all packet traces represented by a workload satisfy the query. For that, we have found efficient ways to abstract time and to encode how packets interact across common network elements such as queues. We then use an SMT solver to reason about packet traces in a workload over a bounded number of time steps. Moreover, we need to define the workload language such that a workload, despite representing bounded packet sequences, can capture the commonality between packet traces that satisfy the query in a concise, intuitive, and generalizable manner. Finally, our search algorithm needs to explore the space of all workloads efficiently and converge to one that satisfies the query in reasonable time. For that, we use a stochastic search algorithm based on Markov Chain Monte Carlo (MCMC) [49], with a cost function that guides the algorithm towards the parts of the search space where there is a higher chance of finding an answer.

We demonstrate the effectiveness of our approach by applying it to packet scheduling algorithms and asking queries about throughput, fairness, and starvation. The fact that packet sequences are bounded, as well as other optimizations in our SMT encoding, enables our framework to analyze each workload in $< 1s$ on average. Moreover, our search algorithm can successfully find workloads that convey high level insights about traffic classes that can experience transient or persistent performance problems in 6-18 minutes.

2 Overview and Motivation

In this section, we use an example packet scheduler inspired from FQ-CoDel [28], the default scheduler for network interfaces in Linux, to demonstrate how formal methods and workload synthesis can be used to reason about performance. FQ-CoDel is a hybrid packet scheduler and active queue management (AQM) algorithm. The scheduler consists of two sets of queues, *new* and *old* queues, that are serviced using a deficit-round-robin (DRR)-like algorithm [50], and each queue is managed with the CoDel AQM algorithm. We focus on the scheduler part of FQ-CoDel as it captures the intricacies of interactions between competing traffic.

The scheduler. The scheduler keeps two lists: a list of the new queues, *new_list*, and a list of the old ones, *old_list*. When a packet comes in, it is first classified (e.g., based on its 5-tuple) and enqueued into a queue. If the queue is not active, i.e., is neither in *new_list* nor *old_list*, it is added to the end of *new_list*. Otherwise, the queue remains in its current list. On dequeue, the scheduler prioritizes queues in *new_list* over those in *old_list* to help short latency-sensitive flows with a few packets not be blocked by longer flows. Specifically, it first looks at *new_list*. If the queue at the head of the list is empty or has already dequeued at least a (configurable) quantum of bytes, it is removed from *new_list* and inserted at the end of *old_list*, and the scheduler goes on to check the next queue in the list. Otherwise, the queue is selected for dequeue. If there are no eligible queues for dequeue in *new_list*, the scheduler repeats the same process on *old_list*. If an old queue is found to be empty during this process, it will be removed from *old_list* and marked as inactive. It can activate again as a new queue when it receives traffic.

The bug. In the correct scheduler, if a new queue becomes empty, it is first demoted to *old_list*, and only if it still stays empty after all the old queues are visited on subsequent dequeues, it will be deactivated. The FQ-CoDel RFC [28] emphasizes the necessity of the demoting step in preventing starvation and warns against directly deactivating a new queue after it becomes empty: "Otherwise, the queue could reappear (the next time a packet arrives for it) before the list of old queues is visited; this can go on indefinitely, even with a small number of active flows, if the flow providing packets to the queue in question transmits at just the right rate."

This is a subtle bug that is difficult to catch with existing approaches for performance analysis. The traffic pattern that reveals the bug consists of a flow sending packets at a very specific rate, which could depend on the number and traffic pattern of other active flows that traverse the scheduler. As such, it is not likely to be part of the common workloads that are tried out in simulation and emulation and can be overlooked in empirical experiments. Similarly, approaches based on queuing theory and network calculus focus on schedulers and workloads that have concise and well-defined mathematical approximations and cannot be readily applied to this specific variation of DRR scheduler or this traffic pattern.

2.1 Using Synthesis to Analyze Performance

Figure 1 shows an overview of how operators can use workload synthesis to reason about the performance of network components like our example scheduler.

Modeling contention points (§3). The operators first spec-

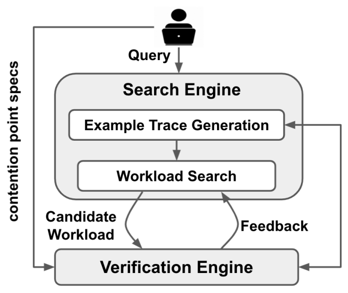

Figure 1: Overview of workload synthesis.

ify which network component(s) they are interested in, and if there are more than one, how those components are connected together. Our verification engine takes this input and generates logical variables and constraints that model these network components and their interactions. Each component is modeled with one or more *queuing modules*, each with n input queues, m output queues, and a processing block in the middle that takes packets from input queues, processes them, and puts the resulting packets into output queues.

Here, we are only interested in a single component, the scheduler, which we model as a queuing module with five input queues and one output queue (Figure 2). In the verification engine, we generate variables and constraints that model these queues and their content for T consecutive time steps, where time advances on every dequeue operation. Every time step, each input queue receives a bounded number of packets from its corresponding flows. The processing block selects one of the input queues for dequeue according to the scheduler's dequeue logic, dequeues a packet from it, and places the packet into the output queue. We model the scheduler's logic with SMT formulas that connect the scheduler state and queue contents at time t to those at time $t+1$.

Performance queries (§4). Next, the operator asks a query about a performance metric of interest such as throughput, latency, or fairness. Since the FQ-CoDel scheduler is supposed to provide fair queuing, the operator can ask whether or not one queue can take more than its share of the bandwidth with the following query:

$$\underbrace{\left(\wedge_{i=1}^{4} \forall t \in [1,T],\, cenq(Q_i,t) \geq t\right)}_{\text{assuming other queues are backlogged}} \rightarrow \underbrace{cdeq(Q_5,T) > 2\lfloor T/5 \rfloor}_{\text{can } Q_5 \text{ get more than its fair share?}}$$

where $cenq(q,t)$ and $cdeq(q,t)$ are the number of packets respectively enqueued into and dequeued from queue q by the end of time step t. The query can be read as "if queues Q_1 to Q_4 are backlogged the entire T time steps, is it possible for Q_5 to dequeue more than twice its fair share (i.e., $\lfloor T/5 \rfloor$)?".

Introducing workloads (§5). At this point, we can ask the verification engine to use an SMT solver like Z3 [6] to find an input *trace* that can satisfy the query. A trace is simply an assignment to the variables that represent how many packets enter each queue in every time step. Figure 2 shows an example trace found by the verification engine that satisfies the query. While the trace does provide a concrete example of a scenario in which Q_5 receives more than its fair share

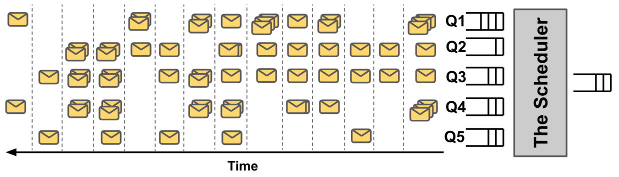

Figure 2: An example trace.

of the bandwidth, it is not necessarily a useful answer to the query. Such a packet trace specifies the ordering and timing of the entry of every single packet and is, therefore, not easy to interpret: is the fact that three packets entered Q_1 in the same time step in the beginning crucial to Q_5 getting a larger share of the bandwidth or is it just an arbitrary choice? Would the query still be satisfied if instead Q_2 had received three packets in the first time step? If the details of timing and ordering in the trace indeed matter, it is not clear if the trace is actually pointing to a subtle but prominent performance problem, or just an extremely rare network scenario which operators are fine to overlook and as such, are not interested in.

Rather than output a single packet trace, our search engine instead synthesizes a *workload* that can describe, in a concise and intuitive manner, a large set of packet traces that can cause performance problems. For the FQ-CoDel example, one such synthesized workload is the following:

$$\begin{aligned}
&\forall t \in [1,6] : cenq(Q_5,t) \leq 1\\
\wedge\, &\forall t \in [7,14] : aipg(Q_5,t) \geq 2\\
\wedge\, &\forall t \in [13,14] : cenq(Q_5,t) \geq 5\\
\wedge\, &\underbrace{\left(\wedge_{i=1}^{4} \forall t \in [1,14],\, cenq(Q_i,t) \geq t\right)}_{\text{backlog assumption from the query}}
\end{aligned}$$

$cenq(q,t)$ is the total number of packets that enter q by $t+1$, and $aipg(q,t)$ is the inter-packet gap between the last packet that has entered q by time $t+1$, and the packet before that. Such an answer is more interpretable as it captures the commonality of a set of traces that satisfy the query. Moreover, the fact that a set of similar packet traces all cause the same performance problem is also a preliminary indication that it represents more than just a rare scenario.

We define workloads as a conjunction of constraints, each specifying a traffic pattern for one or a subset of queues over a period of time. For $T = 14$, the above workload specifies a traffic pattern that will always satisfy the query, i.e., cause Q_5 to dequeue at least five packets when it should not have dequeued more than three. The first constraint states that at most one packet enters Q_5 in the first 6 timesteps, to ensure that unlike the other four queues, Q_5 is not demoted to the list of old queues. After that, the second constraint ensures that Q_5 receives traffic at a specific rate, at most one packet every other time step, so that it is always empty after dequeuing a packet, is wrongfully deactivated due to the bug, is activated as a new queue on the receipt of the next packet, and is prioritized over others for dequeue. The third constraint ensures that Q_5 receives at least five packets by $T = 14$, so it has enough packets to dequeue and satisfy the query. The final constraint

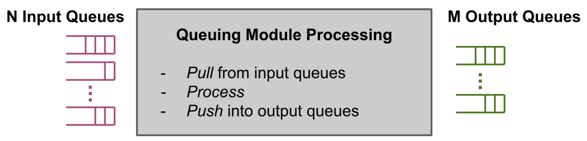

Figure 3: A queueing module.

ensures that the other queues are always backlogged, which comes from the assumption specified in the query.

Synthesizing workloads as answers to queries (§6). The search engine is responsible for generating a workload that satisfies the query. It first uses the verification engine to generate a set of example traces that can guide the search towards finding a suitable workload. Next, inspired by prior work on program and invariant synthesis [47, 49], it starts a stochastic search process based on Markov Chain Monte Carlo (MCMC) that synthesizes a candidate workload and asks the verification engine to verify whether all traces in that workload satisfy the query. If not, the violating trace is returned to the search engine and added to the example traces to help guide finding the next candidate workload. This process repeats until an answer, e.g., the previous workload for our example scheduler, is found, which is then returned to the operator. The operator can either terminate the analysis here, or ask for other different workloads that can satisfy the query.

3 Modeling Contention Points

Queues are an integral part of networks. In fact, networks can be viewed as multiple layers of queues with well-defined functionality in between describing how to deliver data from one layer to the next: When a packet goes from one host to another, it goes from socket buffers, to queues in network interface schedulers in the end-host stack (e.g., Linux qdiscs), to NIC transmit queues on the sender. It then traverses switch input and output queues in the network, and then the NIC, qdisc, and socket buffers on the receiving end. Network contention points, e.g., switches, NICs, and network stacks, heavily rely on queues to decide how to allocate network resources to competing traffic streams. Indeed, performance metrics of a packet stream are significantly affected by the queues its packets traverse and the frequency at which those queues are selected to pass on their traffic. As such, queues are a fundamental part of our model.

Figure 3 shows a simple yet expressive building block for modeling layers of queues: a *queuing module* with $n \geq 1$ input queues, $m \geq 1$ output queues, and a processing block in the middle that takes a batch of packets from a subset of input queues every time step, processes them, and pushes the results into the output queues. To keep track of performance metrics, we designate extra variables per queue for each metric that get updated as packets enter and exit queues, and ask queries about their values for different queues (see §4 for examples).

Composing queuing modules. Queuing modules can be easily composed by feeding the output queues of one module to the input queues of another, as we will show in one of our case studies (Figure 6, §7.4). One could start by modeling only a single bottleneck, e.g., a qdisc, or a NIC/switch scheduler. We can then compose their queuing modules and move on to reasoning about segments or paths in the network (§7). We can even potentially close the loop by designating a queuing module for congestion control algorithms, with one input queue receiving acks and other control packets and one output queue for transmitting data packets. While we have not explored this last direction in this paper, we believe it is a very interesting avenue for future work.

Modeling Time. We define a time step as the time between the dequeue operations of the slowest output queue. That is, time advances when a new dequeue happens. Since our verification engine performs bounded model checking over time, we need to define time at a granularity that ideally allows performance problems to manifest themselves over fewer time steps. Wall-clock time is too fine-grained and, therefore, not a suitable candidate. We observe that when a performance problem occurs, the more interesting corner cases are typically those that limit the maximum number of packets that can enter a queuing module in between dequeues. This makes intuitive sense as if one allows unlimited or a large number of packets to enter a queuing module between dequeues, it is often possible to find "trivial" scenarios that overwhelm the queuing module and create performance problems. Thus, modeling time based on dequeues seems like a good compromise and a natural choice: it is coarser-grained than wall-clock time, and we can easily model the order at which a constrained set of packets enter the queuing module every time step.

Modeling Packets. We model packets as tuples consisting of multiple "metadata" variables. Depending on the query, these variables can include the arrival and departure time of the packet into and out of different queues of interest, flow id, application id, or packet size.

Modeling Queues. A queue is specified with two parameters, its size S and the maximum number of enqueues K allowed at every time step. We define three sets of variables for each queue for every time step t: (1) $enqs[t][1:K]$ consists of K tuples to capture the packets that are sent to the queue at time t. These packets will enter that queue at time $t+1$, (2) $elems[t][1:S]$, consists of S tuples to captures the packets that are inside the queue at time t, and (3) an integer variable $deq\_cnt[t]$ that captures how many packets will be dequeued from this queue at time t. We constrain these variables such that they collectively behave like a FIFO. Finding the right constraints to model FIFOs in a scalable manner is not straightforward, especially when there are multiple enqueues and dequeues per time step. We describe our encoding of FIFOs in SMT in the Appendix.

When two modules are composed, we need extra constraints to move packets from the output queue of one to the input queue of the other. Suppose the first output queue of module A is connected to the first input queue of module B. B decides how many packets to dequeue from $A.out_1$ at time t and sets its $deq\_cnt[t]$ to, say, k. We add constraints to enqueue the first k packets in $A.out_1$ into $B.in_1$, that is, $B.in_1.enqs[t][1:k] = A.out_1.elems[t][1:k]$. Note that this means packets from $A.out_1$ will appear in $B.in_1$ at time $t+1$. If we have a chain of x sequentially-composed queuing modules, it takes x time steps for a packet to traverse the chain. If we only have one chain, or multiple chains of the same length, we can change the constraints such that packets leaving $A.out_1$ at time t are visible in $B.in_1$ in the same time step. That is, analogous to a "run-to-completion" model, each packet is processed by all the queuing modules one-by-one but in the same time step. This will not change the relative ordering of packets across queuing modules when all the paths through the queuing modules have the same length, and reduces the number of time steps needed in the verification engine to analyze workloads. We discuss this optimization in more detail in §7.5.

Modeling packet-processing algorithms. The packet-processing algorithm in a queuing module is responsible for setting the $deq\_cnt[t]$ variables of the module's input queues to denote how many packets will be dequeued from each input queue at time t, and deciding which one of those packets to move to the $enqs[t]$ of the output queues, potentially changing, dropping, or cloning some packets in the process. As long as algorithm's logic can be expressed in SMT, we can plug it into our framework. We envision creating a library that encodes well-known packet-processing algorithms in NICs, switches, and network stacks in SMT. Then, operators just need to specify which algorithm goes into which queuing module and how the queuing modules are composed. In our case studies, we do this encoding manually and leave automatic derivation of SMT encodings from pseudocode or code to future work.

4 Performance Queries

Performance queries are written as logical formulas over one or more performance metric. Operators can define their metrics of interest and have them be tracked for all queues or a subset of queues. They can then ask queries about the value of a certain performance metric for a specific queue, or compare the values of performance metrics between queues.

Performance metrics. A performance metric $m(q,t)$ is a function that computes a value over the packets that have entered or departed the queue q until the end of time step t. Most metrics can be defined as recursive functions over time. For instance, consider a metric d that tracks the maximum delay experienced by packets in a queue. We can define it recursively in the following way:

$$d(q,0) = 0 \quad d(q,t) = max(t-a,\ d(q,t-1))$$
$$\text{where } a = min_{p \in depart(q,t)}\, arrival(p,q)$$

where $depart(q,t)$ is the set of packets that depart from queue q at time t, and $arrival(p,q)$ is packet p's arrival time into q. In defining metrics, one can use simple operations such as addition, subtraction, multiplication with a constant, and taking the maximum and minimum between values.

Queries. Queries are logical formulas over performance metrics. As shown in Figure 4, they ask questions about the value of metrics for a certain queue or compare the value of metrics for different queues over a certain period of time. For instance, using the metric d defined above, we can ask if packets could face significant delay in queue Q with the following query: $\exists t \in [1,T] : d(Q,t) > D$. Moreover, as part of the query, operators can specify base_wl, a workload (formally defined in Figure 4 and §5) that constrains the space of traces the operator is interested in. That is, the final workload return by the search algorithm should be a subset of base_wl. For instance, suppose we want to know, assuming a minimum input rate R for a queue Q, whether it will transmit fewer than K packets during T time steps. We can ask if

$$\underbrace{\forall t \in [1,T] : cenq(Q,t) \geq R \cdot t}_{\text{base\_wl}} \rightarrow \exists t \in [T,T] : cdeq(Q,t) < K$$

where $cenq(q,t)$ and $cdeq(q,t)$ track the total number of packets that have entered and exited q by end of t. Building on the previous query, we can also investigate fairness between two queues by assuming minimum input rates for both and comparing the number of packets they transmit:

$$\forall t \in [1,T] : cenq(Q_1,t) \geq R_1 \cdot t \ \wedge\ \forall t \in [1,T] : cenq(Q_2,t) \geq R_2 \cdot t$$
$$\rightarrow \exists t \in [T,T] : cdeq(Q_1,t) - cdeq(Q_2,t) \geq T/2$$

Thus, queries can ask about many performance metrics, including latency, throughput, fairness, and starvation.

$$\begin{array}{l|l}
\mathsf{qry} := \mathsf{wl} \rightarrow \mathsf{tr} : \mathsf{lhs} \oplus \mathsf{rhs} & \mathsf{wl} := \mathsf{true} \mid \mathsf{con} \wedge \mathsf{wl} \\
\mathsf{tr} := \{\forall \mid \exists\} t \in [T_1, T_2] & \mathsf{con} := \forall t \in [T_1, T_2] : \mathsf{lhs} \oplus \mathsf{rhs} \\
\mathsf{lhs} := m(Q,t) \mid m(Q_1,t) - m(Q_2,t) & \mathsf{lhs} := m(Q,t) \mid \Sigma_{q \in \{Q_i\}}\, m(q,t) \\
\mathsf{rhs} := C \cdot t \mid C & \mathsf{rhs} := C \cdot t \mid C
\end{array}$$

Figure 4: Syntax for queries (§4) and workloads (§5).

5 The Workload Language

Workloads are also specified as logical formulas over a set of metrics. A workload is a set of constraints, each specifying the traffic pattern for a subset of queues over a period of time. Workloads only constrain *input queues*, i.e., queues that receive packets from "outside" as opposed to another queuing module, to help operators analyze how a contention point will perform under different classes of external traffic. More formally, as shown in Figure 4, a workload wl is a conjunction of constraints (con). Each con is of the form $\forall t \in [T_1, T_2] : \mathsf{lhs} \oplus \mathsf{rhs}$, where T_1 and T_2 are integers constraining the interval of time for which this constraint governs the traffic of its corresponding queues, lhs is either a metric for one queue, or the aggregate of a metric over a subset of queues, $\oplus$ is a comparison operator that shows how the lhs will be constrained by the rhs, which time, or a constant.

Workloads can describe sets of traces in a concise and intuitive manner. Consider the single-constraint workload: $\forall t \in [1,10] : cenq(Q_1,t) \geq t$. Any trace that satisfies this constraint, that is, sends at least t packets into Q_1 by time t and does not leave the queue idle, is part of this workload. It does not matter if the traffic comes in one packet at a time, or if 10 packets all come in at time step 1, or if 5 packets enter in the beginning, and 5 more at time step 5. That is, workloads can abstract away small details of packet traces as long as a higher-level property, as specified with a metric, is satisfied.

Workload metrics. The search algorithm explores the space of all workloads to find one that satisfies the query. When synthesizing candidate workloads for the verification engine to check, it decides how many constraints to include in the workload, and what metrics and queues to include in each constraint (§6). While we leave the metrics that can be used in queries unconstrained, deciding the set of metrics that can be used in synthesizing workloads requires careful consideration. We want the set to be small to keep the search space tractable but expressive to enable specifying common kinds of workloads in a concise and intuitive manner.

We define our workloads over two metrics: (1) cumulative enqueues, or $cenq(q,t)$, the total number of packets enter q by the end of time step t, and (2) arrival inter-packet gap, or $aipg(q,t)$, the inter-packet gap between the last packet the enters q by time t, and the packet before that. Although small, this is a quite expressive set. *cenq* only constrains the total number of packets entering the queue, independent of the exact time they arrive. As such, it can abstract away the low-level timing details of traces when not important for a query. *aipg*, on the other hand, constrains the gap between packets and the pace at which they arrive. So, it can capture low-level timing details if necessary in answering the query.

We believe that together, they create a good balance in capturing the commonalities of traces that satisfy a query into a concise and intuitive workload, abstracting away unnecessary details and including necessary ones. We view it as a reasonable starting point, and hope that as using workload synthesis for performance analysis evolves, the set of metrics will mature as well. In fact, our search algorithm is parametrized over the set of metrics and any metric that can be encoded in SMT can be easily added to our search algorithm.

6 Synthesizing Answers

Given a query, the search engine uses a guided randomized search over the space of workloads to find one that satisfies the query. The search algorithm, shown in Algorithm 1, is based on the Metropolis Hastings Markov Chain Monte Carlo (MCMC) sampler, which combines random walks with hill climbing and has been successfully used for synthesizing optimized programs and loop invariants [47, 49, 54]. Starting from an initial workload $\mathsf{wl} = \mathsf{true}$ (line 3), which imposes no constraints on the input queues, the search algorithm asks the verification engine to verify whether this workload is valid and satisfies the query, i.e., whether all the traces in the workload satisfy the query (line 6). If that is the case, the search engine returns the workload as the answer to the query (line 7). If not, using the feedback from the verification engine, the search

Algorithm 1: Workload synthesis search.

```
   Input: User query (qry) from Figure 4.
   Output: Workload formula (wl) from Figure 4.
1  Procedure Search(qry)
2      if (not feasibleBaseWorkload(qry)) return BadQuery
3      wl = true; G = goodSet(qry); B = badSet(qry)
4      $c_1 = \mathrm{cost}(wl, G, B)$
5      while (true) do
6          (found, bad_trace) = verify(wl, qry)
7          if (found) break else B.insert(bad_trace)
8          op = randomOperation()
9          next_wl = wl.apply(op)
10         $c_2 = \mathrm{cost}(next\_wl, G, B)$
11         if ($c_1 > c_2$) then wl = next_wl; $c_1 = c_2$
12         else if ($e^{-\lambda \cdot (c_2 - c_1)} > \mathrm{rand}()$) wl = next_wl; $c_1 = c_2$
13     shrink(wl); broaden(wl); return wl
```

algorithm moves on to synthesize and try another candidate workload until a suitable workload is found (lines 8-12).

6.1 Verifying workloads

Given a workload wl, and a query $\mathsf{base\_wl} \rightarrow \mathsf{qry}$, the verification engine uses an SMT solver [6] to check whether the following formula is satisfiable: $\mathsf{model} \wedge \mathsf{base\_wl} \wedge \mathsf{wl} \wedge \neg\mathsf{qry}$, where model is the logical encoding of the queueing modules the operator is interested in (§3). If the above formula is satisfiable, there is at least one valid trace that is both in $\mathsf{base\_wl}$ (the space of traces the operators are interested (§4)) and wl, but does not satisfy the query. In that case, the trace is returned to the search algorithm to guide the synthesis of the next candidate workload.

If the formula is not satisfiable, it is either the case that (1) $\mathsf{base\_wl}$ or wl or their combination is *infeasible*, meaning that no trace can satisfy their constraints and they actually represent the empty set, or that (2) there are no traces in $\mathsf{base\_wl} \wedge \mathsf{wl}$ that do not satisfy the query. It is only in case (2) that wl is a valid and non-trivial answer to the query. We need to distinguish these two cases to ensure that only feasible workloads are returned as answers. For that, the search engine asks the verification engine whether $\mathsf{model} \wedge \mathsf{base\_wl}$ is satisfiable (line 2). If not, the space of traces specified by $\mathsf{base\_wl}$ is actually empty. So, the search engine does not start the search and notifies the user to modify $\mathsf{base\_wl}$. Moreover, when verifying a candidate workload wl (verify on line 6), the verification engine also checks whether $\mathsf{model} \wedge \mathsf{base\_wl} \wedge \mathsf{wl}$ is satisfiable. If not, the fact that wl is infeasible is also returned as feedback to guide the selection of the next candidate.

6.2 Generating the next candidate

If a candidate workload is not the final answer, either because it is infeasible or it contains a trace that does not satisfy the query, the search algorithm synthesizes another candidate workload to try. The next candidate is a mutation of the previous one. Suppose the previous candidate is $\mathsf{wl} = \wedge_{i=1}^{k} \mathsf{con}_i$. The search algorithm chooses one of the following operations at random (line 8), and applies it to wl (line 9), to obtain $\mathsf{next\_wl}$:

- **Add** a new random constraint con, so $\mathsf{next\_wl} = \mathsf{con} \wedge \mathsf{wl}$.
- **Remove** a random constraint con_j from wl. That is, $\mathsf{next\_wl} = (\wedge_{i=1}^{j-1} \mathsf{con}_i) \wedge (\wedge_{i=j+1}^{k} \mathsf{con}_i)$.
- **Modify** a random constraint con_j. Suppose $\mathsf{con}_j = \forall t \in [T_{1_j}, T_{2_j}] : \mathsf{lhs}_j \oplus_j \mathsf{rhs}_j$. The search algorithm randomly picks whether to change one of T_{1_j}, T_{2_j}, lhs_j, $\oplus_j$, or rhs_j to obtain con'_j, so that $\mathsf{next\_wl} = (\wedge_{i=1}^{j-1} \mathsf{con}_i) \wedge \mathsf{con}'_j \wedge (\wedge_{i=j+1}^{k} \mathsf{con}_i)$.

These operations are motivated by prior work that has empirically shown MCMC to work well with a mixture of major (add and remove) and minor (modify) changes to the current candidate to obtain the next one [47, 49].

Next, the search algorithm uses a cost function (§6.3) to decide whether it is "worth" transitioning to $\mathsf{next\_wl}$ and try it out. If $\mathsf{next\_wl}$ has a lower cost compared to wl, the algorithm makes the transition and $\mathsf{next\_wl}$ becomes the current candidate (line 11). If $\mathsf{next\_wl}$ has a higher cost, the algorithm may still choose, probabilistically, to make the transition not to get stuck in local minima. Suppose c_{wl} and $c_{\mathsf{next\_wl}}$ are the costs of the previous and next candidate, respectively. If $c_{\mathsf{next\_wl}} > c_{\mathsf{wl}}$, the algorithm will make the transition to $\mathsf{next\_wl}$ with probability $e^{-\lambda(c_{\mathsf{next\_wl}} - c_{\mathsf{wl}})}$ (line 12). That is, the probability of transitioning from a candidate with a lower cost to one with a higher cost is proportional to the difference in their costs. If the algorithm chooses not to transition to $\mathsf{next\_wl}$, it picks a new candidate workload and repeats this process. Once it transitions into a new candidate workload, it asks the verification engine to check whether the workload satisfies the query.

6.3 The Cost Function

The cost function guides the search towards a suitable workload and is, therefore, critical in the efficiency of the search algorithm. Workloads represent sets of traces, and we want the search algorithm to find one such that all the traces in that workload satisfy the query. Intuitively, a good cost function should (1) favor workloads when they have a large number of packet traces that satisfy the query, and (2) penalize those that include traces that do not satisfy the query.

Inspired by prior work [49], we quantify these criteria into a cost function using example traces. We create two sets of example traces before starting the search (line 3): a set of *good* example traces (G), all of which satisfy the query, and a set of *bad* examples (B), none of which satisfy the query. Suppose $\mathsf{match}(\mathsf{wl}, \mathsf{E})$ is the number of traces in E that are also in wl. The cost function is then defined as: $\mathsf{cost_E}(\mathsf{wl}, \mathsf{G}, \mathsf{B}) = \mathsf{match}(\mathsf{wl}, \mathsf{B}) - \mathsf{match}(\mathsf{wl}, \mathsf{G})$. Recall that if a workload is feasible but does not satisfy the query, the verification engine returns a trace in that workload but does not satisfy the query as feedback. These traces are added to B as the search goes on, further refining the cost function (line 7).

Note that even if a workload matches some bad examples, it can still have a low cost and get selected as the next candidate. This is acceptable because the search algorithm may need to go through "obviously" bad workloads to explore different regions of the search space and find the answer. Also, example traces are used only to guide the search; each candidate workload is *verified* in the verification engine to ensure that every trace represented by the workload satisfies the query.

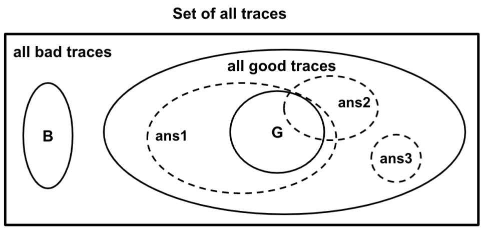

Figure 5: Relationship between answer workloads and G and B.

Breaking "ties". $\mathsf{cost_E}$ effectively guides the search toward workloads that satisfy the query, specifically those that include most of the traces in the G. But there could be more than such workload. To ensure interpretability, we define another cost function $\mathsf{cost_S}(\mathsf{wl})$ that favors *concise* workloads, i.e., those the have fewer constraints, and constrain fewer queues over longer, less fragmented periods of time. Our final cost function is a weighted combination of $\mathsf{cost_E}(\mathsf{wl})$ and $\mathsf{cost_S}(\mathsf{wl})$:

$$\mathsf{cost}(\mathsf{wl}) = \mathsf{C_E} \cdot \mathsf{cost_E}(\mathsf{wl}) + \mathsf{C_S} \cdot \mathsf{cost_S}(\mathsf{wl})$$

We set $\mathsf{C_E} > \mathsf{C_S}$ so that in the beginning, the search algorithm probes the space of workloads for *any* answer that can satisfy many good examples and no bad ones. Once the algorithm has reduced $\mathsf{cost_E}(\mathsf{wl})$ and is in the "right" part of the space, $\mathsf{cost_S}(\mathsf{wl})$ helps direct the search towards a more concise answer. The definition of $\mathsf{cost_S}(wl)$ can be found in the appendix.

6.4 Generating The Example Sets

Before the search starts, the search engine communicates with the verification engine to generate traces for G and B We formally define a trace eg as a 2D array that concretely specifies how many packets enter each queue at each time step. For example, if $eg[Q_1][5] = 3$, it means that in this trace, Q_1 receives 3 packets at time 5.

The bad examples set (B)**.** The ith bad example is a trace that satisfies the following formula:

$$\mathsf{model} \wedge \mathsf{base\_wl} \wedge \neg\mathsf{qry} \wedge \neg(\vee_{j=1}^{i-1} \mathsf{eg_i} \neq \mathsf{eg}_j) \wedge \mathsf{local\_mods}_i$$

Having $\mathsf{model} \wedge \mathsf{base\_wl} \wedge \neg\mathsf{qry}$ in the formula ensures that the trace does not satisfy the query and $\neg(\vee_{j=1}^{i-1} \mathsf{eg_i} \neq \mathsf{eg}_j)$ ensures it is different from the previous $i-1$ traces. $\mathsf{local\_mods}_i$ ensures that there is variety across the traces in B so that the search algorithm can prune the search space faster. We pick P random points $(q_1, t_1), \cdots, (q_P, t_P)$ in the $(i-1)$th trace and P random integers $v_1, \cdots, v_P$ such that $\mathsf{eg}_{i-1}[q_j][t_j] \neq v_j$. Then, we define $\mathsf{local\_mods}_i = \wedge_{j=0}^{P} \mathsf{eg}_i[q_j][t_j] = v_j$. So, the ith trace is different from the previous one in at least P points. If no such trace could be found after two tries, we decrement P and retry until a new trace is found.

The good examples set (G)**.** Generating G is more complicated. To see why, consider the diagram in Figure 5 and an arbitrary workload ans that satisfies the query. No matter how we choose B, ans cannot not include any of its traces. So, by minimizing match(wl, B) in the cost function, the search algorithm is always moving in the direction of an answer. However, depending on how we pick G, ans may include all (ans_1 in Figure 5), a subset (ans_2), or none (ans_3) of G's traces. There may not even exist a workload like ans_1 that can express all the traces in G without including any bad traces. So, if we do not pick G's traces carefully, by maximizing match(wl, G), the algorithm may repeatedly be directed towards a part of the search space where there are no suitable answers.

Intuitively, we want to pick traces for G that are (1) not radically different from each other, so that the majority of them can be represented with a single workload, and (2) not too similar or narrow, so that the workload found by the search algorithm is sufficiently general. To do so, the search engine first asks the verification engine to create a *base* good example trace that it will use to as the basis of generating the rest. Specifically, it asks for a trace eg_0 that satisfies model $\wedge$ base_wl $\wedge$ qry while using max-SMT to minimize the following criteria in the specified order:

1. **Total number of queues with traffic**, i.e., $\Sigma_q I_q$, where $I_q = 1$ if $\Sigma_t eg_0[q][t] = 0$, and is zero otherwise. So if there is a set of traces with traffic in, say, Q_1, Q_2, and Q_3, that satisfy the query, and another set with traffic in, say, Q_2, Q_4, and Q_5, we include example traces from one and not both, to keep the search more focused.

2. **Total number of time steps queues do not receive traffic**, i.e., $\Sigma_{q,t} I_{q,t}$, where $I_{q,t} = 1$ if $eg_0[q][t] = 0$ and is zero otherwise. This means that in the base trace, every queue receives at least one packet every time step as long as it is "harmless", that is, it does not stop the trace from satisfying the query. This smooth background traffic in the base trace can be randomly changed in the rest of the traces to ensure diversity in the good examples set.

3. **Total amount of traffic in the trace**, i.e., $\Sigma_{q,t} eg_0[q][t]$. Given the above optimization criteria, this ensures that the background traffic is not flooding the contention point with too many packets in each time step and only allows more than one packet per time step in the base example trace if it is necessary for satisfying the query.

The rest of G's traces are generated from eg_0. Similar to generating B, the search engine asks for a trace that satisfies the query, is not any of the traces generated before, and is different from the previous trace in at least P random places. There are two differences: for the ith trace eg_i, we (1) add extra constraints so that the queues that have no traffic in eg_0 (see optimization criteria 1) have no traffic in eg_i either, and (2) minimize the "distance" between eg_i and eg_0, i.e., minimize $\Sigma_{q,t} d_{q,t}$, where $d_{q,t} = 1$ if $eg_i[q][t] \neq eg_0[q][t]$ and zero otherwise.

6.5 Generating the Final Workload

Once the search engine finds a workload wl that satisfies the query, it creates a new workload ans_wl that includes all the constraints from wl and base_wl. It then performs "workload shrinking" (Algorithm 1, line 13): It tries removing every constraint in ans_wl one at a time and check if it still satisfies the query. This helps remove any constraint that may have been added to base_wl during example generation (§6.4) or to wl during the search that is not necessary for satisfying the query. Then, it tries "workload broadening". For a queue Q_i that is not in ans_wl and a constraint con in, if con's left hand side is $m(Q_j, t)$, it is changed to $\Sigma_{q \in \{Q_i, Q_j\}}$, and if it is $\Sigma_{q \in \{Q_j\}}$, it is changed to $\Sigma_{q \in \{Q_j\} \cup Q_i}$. If the workload still satisfies the query, this helps include even more traces in the workload. The updated ans_wl is then returned to the user as the final answer. The user can either terminate the analysis here, or ask for another workload that is not ans_wl.

6.6 Optimizations

We have employed several optimizations to Algorithm 1 to make the synthesis process more efficient.

Reducing the search space. When generating G, we create a base trace eg_0 that satisfies the query while minimizing the number of queues that have traffic (among optimizing other criteria, see §6.4). If a queue q has no traffic in eg_0, it means that as long as the other "non-idle" queues have traffic, its traffic is either not important or has to stay zero for satisfying the query. So, we temporarily add $\forall t \in [1, T] : cenq(q, t) = 0$ to base_wl and only look for workloads that constrain the rest of the queues during search. This reduces the space of workloads the search algorithm needs to explore. When a workload that satisfies the query is found, the workload minimization step (§6.5) removes these constraints if not necessary. Moreover, our workload language allows for easy mutation of workloads with simple operations during search to generate new candidates. So, it is possible for a workload's mutation to represent the same set of traces while being syntactically different. We perform several checks to detect such "duplicate" workloads and avoid generating them as candidates, reducing the space of workloads the search algorithm needs to explore.

Reducing calls to the verification engine. Each call to the verification engine can be quite expensive as it it checks the satisfiability of non-trivial formulas. So, we try to avoid calling the verification engine whenever possible, both during the search and in the post-processing steps. If the search algorithm selects a candidate workload that matches a trace in B, it can move on to finding the next candidate without consulting with the verification engine, as it already knows that the current candidate includes a trace that does not satisfy the query. Note that the search algorithm selects these candidates despite that fact that they match traces in B as they could help it explore different regions of the search space. Similarly, if a workload is rejected because it is infeasible (§6.1), the search engine will keep track of it and avoid calling the verification

	Prio	RR	FQ-Codel	Comp
Queuing Modules	1	1	1	11
Queues	5	6	7	29
Boolean Vars ($\times$1000)	0.84	0.15	2.6	10.6
Integer Vars ($\times$1000)	0.64	1.1	1.9	7.3
Constraints ($\times$1000)	7.2	13	2.2	93.8
Max timesteps	7	10	14	10

Table 1: Case study statistics

engine if that workload comes up in the future.

Escaping local minima. As with any search that optimizes an objective, there is a chance that our search algorithm hits a local minimum and cannot find any "neighboring" candidates with a lower cost. To avoid getting stuck in local minima, the search algorithm keeps track of its progress, i.e., the difference between the cost of the previous candidate workload and the next one. If the progress is below a threshold for a number of rounds, it "looks ahead" a couple of hops when generating that next candidate. That is, instead of applying one of the moves in §6.2 to the current candidate, it applies a few of them in sequence to generate the next workload. If it still cannot make enough progress in another several rounds, we restart the search process from a workload with no constraints.

Other optimizations. Instead of only applying one of the operations in §6.2 and generating one candidate workload, we apply all of them one at a time, generate a set of candidates, and pick one randomly from the ones with the lowest cost. This helps the algorithm explore the lower-cost regions of the search space earlier. Moreover, we introduce a new operation, replace, which replaces a randomly-chosen constraint in the workload with a new random constraint. Replace is equivalent to a remove followed by an add, but it helps the algorithm to generate a more diverse set of candidates faster. Finally, if the search algorithm selects a candidate workload that is infeasible, adding or modifying constraints in the workload to generate the next one are likely to yield another infeasible candidate. So, the algorithm backtracks to the last known feasible candidate and continues from that point.

7 Case Studies: Packet Scheduling

We have prototyped our techniques in a tool we call AutoPerf in ~10K lines of C++ code. We use the Z3 theorem prover [6] in the verification engine for checking the satisfiability of SMT formulas. In this section, we describe our experience in using AutoPerf to analyze packet scheduling algorithms. Why packet scheduling? Packet schedulers are ubiquitous in network components. When multiple traffic streams need to share the same link, be it the NIC, a switch egress port, or a virtual interface, often a packet scheduler determines the order and time at which packets from each traffic stream are transmitted over the link. By using AutoPerf to analyze packet schedulers, we can evaluate how effective our techniques are in capturing the intricate interactions between packets of competing traffic in a concise and intuitive manner.

We discuss four case studies (Table 1): the first two analyze priority-based and round-robin schedulers, which are simple and common packet schedulers that we use as sanity checks to exercise different aspects of our techniques. Next, we discuss the FQ-CoDel scheduler introduced in §2. Our final case study is inspired from Loom [52] and is a composition of 11 queuing modules that model the interaction between schedulers on the host and a multi-queue NIC. Across the four case studies, we ask queries about starvation, throughput, and fairness.

In these case studies, we aim to explore (1) *expressiveness*, i.e., whether our workload language can express a wide range of workloads in answering queries, (2) *interpretability*, whether the final workloads are concise, intuitive, and interpretable, and (3) *tractability*, i.e., whether workloads are generated in a reasonable human timescales (i.e., minutes).

7.1 Priority-Based Scheduling

Our priority scheduler is a single queuing module with four input queues, $Q_1, \cdots, Q_4$, and one output queue. Q_i has a higher priority than Q_j if $i < j$. Every time step, the scheduler picks the highest priority non-empty input queue, dequeues one packet from it, and places the packet in the output queue. In a strict priority scheduler, lower priority queues may get starved, which we quantify using the metric $blocked(q,t)$:

$$\begin{aligned} blocked(q,0) &= \text{if } (\neg q.empty[0] \wedge q.deq\_cnt = 0) \text{ then } 1 \text{ else } 0 \\ blocked(q,t) &= \text{if } (\neg q.empty[t] \wedge q.deq\_cnt = 0) \\ &\qquad \text{then } blocked(q,t-1)+1 \text{ else } 0 \end{aligned}$$

That is, $blocked(q,t)$ is the number of consecutive time steps that q has had packets but not chosen for transmission. We then ask $\exists t \in [1,T] : blocked(Q_3,t) > 5$. For $T \leq 10$, it means Q_3 is continuously blocked for more than half the time. We start from $T = 5$ and increment T until the verification engine can find a satisfying trace in the example generation phase, which is $T = 7$. Then, the search algorithm finds

$$\begin{aligned} &\forall t \in [1,7] : \Sigma_{q \in \{Q_1,Q_2\}} cenq(q,t) \geq t \\ &\wedge \forall t \in [1,7] : cenq(Q_3,t) \geq 1. \end{aligned}$$

The first constraint specifies that there should be consistent flow of traffic into either of Q_1 and Q_2, which have a higher priority than Q_3, and the second constraint specifies that Q_3 should at least have one packet to be considered blocked. While the answer is not surprising, it demonstrates AutoPerf's effectiveness to successfully abstract away the details of which of the higher priority queues receives a packet at exactly what time step and only capture the necessary details.

7.2 Round-Robin Scheduling

Our round-robin scheduler is a single queuing module with five input queues and one output queue. The input queues are serviced in a round-robin fashion (independent of packet size, see §8). If every queue receives steady traffic over a time period T, each should be selected for dequeue at least $T/5$ times. So, when we ask if Q_3 can dequeue more packets than

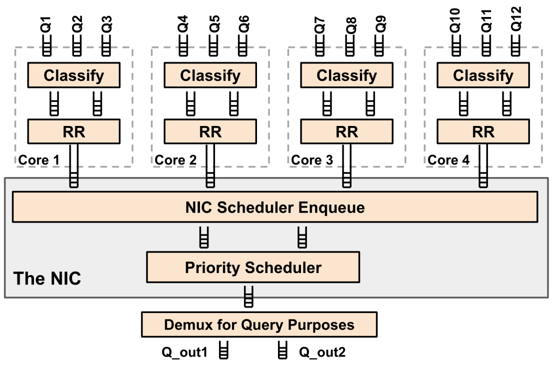

Figure 6: Setup for case study.

Q_2 with the following query

$$\text{base\_wl} \rightarrow \forall t \in [10,10] : cdeq(Q_3,t) - cdeq(Q_2,t) \geq 3$$

where $\text{base\_wl} = \wedge_{i=1}^{5} \forall t \in [1,10] : cenq(Q_i,t) \geq t$, the verification engine cannot find any traces that satisfy the query. Now suppose we relax base_wl to restrict the average rate of traffic into queues (as opposed to every time step), i.e., $\text{base\_wl} = \left(\wedge_{i=1}^{5} \forall t \in [5,5] : cenq(Q_i,t) \geq 4\right) \wedge \left(\wedge_{i=1}^{5} \forall t \in [10,10] : cenq(Q_i,t) \geq 8\right)$, AutoPerf finds

$$\forall t \in [1,4] : \Sigma_{q\in\{Q_1,Q_2,Q_4,Q_5\}} cenq(q,t) \leq 0$$
$$\wedge \forall t \in [1,4] : cenq(Q_3,t) \geq t \wedge \text{base\_wl},$$

which describes a workload in which (1) all of the queues except Q_3 receive no packets in the first four time steps and receive a burst at time 5, while (2) Q_3 continuously receives traffic. In such a scenario, while the average input rate of all queues is the same, other queues temporarily fall behind Q_3 in terms of dequeues due to the burstiness of their traffic.

7.3 The FQ-Codel Scheduler

This case study analyze the scheduler inspired from the FQ-Codel qdisc [28]. It is a single queuing module with five input queues and one output queue. The scheduler's logic, the query, and the workload are described in §2 as our motivating example. Here, we only report that the same workload was overwhelmingly returned as the answer across all ten runs.

7.4 Composing Host and NIC Schedulers

Modern NICs expose multiple transmit queues to the host, so that CPU cores can concurrently send traffic to the NIC, each through a dedicated transmit queue [2, 46, 52]. This provides significant performance benefits but makes it difficult to enforce policies about how applications on the same host should share network resources. Prior work [52] demonstrates this using an example, which we analyze in this case study. Suppose two tenants reside on a server whose CPU cores have dedicated queues to the NIC. Tenant 1 runs spark [1], and tenant 2 runs both spark and memcached [3]. All applications are multi-threaded and can use all the cores. We want to ensure that (1) the two tenants fairly share the network bandwidth, and (2) tenant 2's memcached traffic is prioritized over its own spark traffic. One option, described in [52], is to use software packet schedulers (e.g., Linux qdiscs) to enforce fair sharing between the tenants on the host, and a priority scheduler on the NIC to enforce prioritization of memcached traffic. Note that to avoid overhead, software schedulers only enforce policies per CPU core not across cores.

Figure 6 shows how we model this in AutoPerf for 4 CPU cores. There is one input queue for traffic from each application on each core. That is, $Q_{3(i-1)+1}$, where i is the core number, are for tenant 1's spark, $Q_{3(i-1)+2}$ for tenant 2's spark, and Q_{3i} for tenant 2's memcached traffic. For instance, if tenant 1's spark application only runs on cores 2 and 3, its traffic enters through Q_4 and Q_7, and Q_1 and Q_{10} stay empty. For each core, a queuing module first classifies traffic from that core's three input queues into two output queues, one for each tenant. Then, a round-robin scheduler (§7.2) shares bandwidth between the two tenants into the NIC's transmit queue. The NIC has a module, with four transmit queues as input, that classifies traffic from the cores into two output queues, one for spark and one for memcached traffic. Finally, a priority scheduler (§7.1) that prioritizes memcached traffic decides what packet to send out of the NIC. We also add a "dummy" module that takes the output from the NIC and "demultiplexes" it into one output queue for tenant 1 (Q_{out1}) and one for tenant 2 (Q_{out2}), which we use in our queries.

Since the final scheduler always prioritizes memcached traffic over spark, it is easy to see how the second half of the policy is always enforced. In our query, we ask whether tenant 1 and tenant 2 will get equal access to the NIC output link:

$$\text{base\_wl} \rightarrow \forall t \in [10,10] : \ cdeq(Q_{out2},t) - cdeq(Q_{out1},t) \geq 3$$
$$\text{base\_wl} = (\forall t \in [1,10] : \ \Sigma_{q\in tenant1\_qs}\, cenq(q,t) \geq t) \wedge$$
$$(\forall t \in [1,10] : \Sigma_{q\in tenant2\_qs}\, cenq(q,t) \geq t)$$

$tenant1\_qs$ are $Q_{3(i-1)+1}$ $(1 \leq i \leq 4)$, which carry tenant 1's spark traffic, and the rest of the queues are in $tenant2\_qs$. That is, in base_wl, we are ensure that both tenants receive a steady stream of packets. AutoPerf finds the following workload

$$\forall t \in [1,10] : \ cenq(Q_4,t) \geq t$$
$$\wedge \forall t \in [1,10] : \ cenq(Q_9,t) \geq t$$
$$\wedge \left(\wedge_{i \in rest} \forall t \in [1,10] : \ cenq(Q_i,t) \leq 0\right).$$

where $rest$ is $\{1,\cdots,3,5,\cdots,8,10,\cdots 12\}$. That is, if tenant 2's memcached runs on core 3 and tenant 1's spark on core 2, they will only compete in the priority scheduler in the NIC, where memcached traffic is prioritized over spark traffic, and tenant 2 is favored to access the link. The same phenomena can happen as long as tenant 2's memcached traffic and tenant1's spark traffic come from two different cores.

Next, we modify the base workload to see if the problem still exists if there is no memcached traffic. We add $\bigwedge_{i\in 3,6,9} cenq(Q_i,t) = 0$ to the base workload and repeat the query. This time, we get the following workload as answer:

$$\forall t \in [1,10] : cenq(Q_8,t) \geq t$$
$$\wedge \forall t \in [1,10] : \ cenq(Q_{11},t) \geq t$$

Phases		Prio	RR	FQ-C	Comp
Example Generation	Generating base trace (s)	0.3	1.0	3.1	70.3
	Generating good set (G) (s)	6.8	309.5	346.1	320.5
	Generating bad set (B) (s)	1.2	3.0	7.1	178.5
Verification Engine	Verifying workload (avg) (s)	0.02	0.02	0.11	0.65
	Verifying workload (max) (s)	0.04	0.13	0.86	7.60
	Verifying query (avg) (s)	0.03	0.04	0.10	0.81
	Verifying query (max) (s)	0.06	0.18	0.73	9.90
Search (see §6.6)	# Rounds	65	268	769	361
	# Rounds w.o/verification	39	107	537	131
	# Rounds w/"look-ahead"	1	11	44	20
	Total search time (s)	2.4	58.8	222.9	460.7
Post Processing	Workload shrinking (s)	0.2	0.0	0.9	107.2
	Workload broadening (s)	1.0	0.0	0.1	44.6
Total Time (min.)		0.2	6.2	9.6	18.5

Table 2: Statistics from running AutoPerf 10 times for each case study. **Paramters:** Queue parameters are $S = 10$ and $K = 4$ packets. For example sets, $|G| = |B| = 50$, and $P = min(10, \frac{trace\_size}{5})$. During search, the maximum number of constraints in the workload is set to twice the number of input queues, threshold for slow progress is $0.03 \cdot max(\mathsf{cost}_\mathsf{E})$, $\frac{\mathsf{C_E}}{\mathsf{C_S}} = 10$, and number of rounds of slow progress tolerated before look-ahead and restart is 10 and 20 respectively.

$$\land \forall t \in [1,10] : \Sigma_{i\in\{2,5\}} cenq(Q_i,t) \geq t$$
$$\land \forall t \in [1,10] : cenq(Q_{10},t) \geq t$$
$$\land (\land_{i \in rest} \forall t \in [1,10] : cenq(Q_i,t) \leq 0).$$

where $rest = \{1,3,4,6,7,9,12\}$. Here tenant 2's spark runs on all cores and tenant 1's spark only on core 4. The first three constraints ensure that there are three different concurrent streams of traffic from tenant 2's spark. Since there is only one stream of spark traffic for tenant 1, that is, because the spark flows are not uniformly spread across cores, tenant 1 gets access to the link less frequently than tenant 2. Both workloads are consistent with the empirical results in [52].

7.5 Tractability

In this section, we discuss our main insights from case studies about the tractability of our approach. Table 2 summarizes statistic about different phases involved in workload synthesis for our four case studies, *Prio* (§7.1), *RR* (§7.2), *FQ-CoDel* (§7.3), and *Comp* (§7.4). All statistics are averaged across 10 runs (parameters described in Table 2) on a server with 4-socket NUMA-enabled Intel Xeon Gold 6128 3.4GHz CPU with 6 cores per socket. For Comp, we leave out the statistics about the first query, because in its base example trace, all queues except for Q_4 and Q_9 have zero traffic. So, we add constraints to base_wl to keep them empty throughout the search (§6.4). At this point, the base workload actually satisfies the query, so it is returned to the user after post-processing and without going through the search. For this reason, we only include results for the second query.

Generating (good) examples is expensive. Generating the good example traces is a time-consuming step, taking ~ 7 to 346 seconds. This is not surprising: when generating trace eg_i and starting from $p = P$, we pick p random places in eg_{i-1},

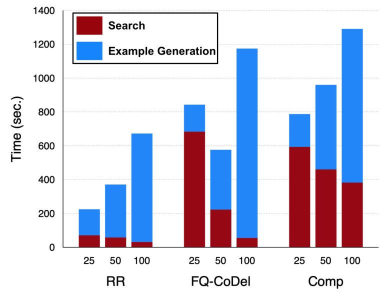

Figure 7: Search and example generations times for $|\mathsf{G}| = |\mathsf{B}| = 25$, 50, and 100. Prio results are not shown as the total time was less then $20s$ for Prio and would not be visible in the plot.

and constrain eg_i to have different randomly chosen values in those p places. If no trace is found in two tries, we decrement p and try again. Each time, we also ask the verification engine to minimize the distance between eg_i and the base trace. The more such calls to the verification engine, either because of the sheer size of G, or because we have to retry a lot for each trace, the longer it will take to generate G.

For Prio and Comp, we often get a good trace on the first try but for RR and FQ-CoDel, generating a good trace takes an average of 16 and 6 tries, respectively. This is because for RR and FQ-Codel, they query is more sensitive to the timing of packets in the trace, RR needing a specific kind of burstiness and FQ-CoDel a specific rate, to be satisfied. As such, making p random modifications to eg_{i-1} for larger values of p makes it improbable to satisfy the query for eg_i, increasing the number of retries. By default, $P = min(10, \frac{trace\_size}{5})$, which is 10 for RR and FQ-CoDel, 8 for Comp, and 3 for Prio. Instead, when generating trace i, we can set P to the moving average of the p that has worked for the previous $(i-1)$ traces. For RR and FQ-Codel, the average call time to the verification engine for each try is at most about a second, so we expect this optimization to considerably reduce the time for generating G.

Another option is to reduce $|\mathsf{G}|$, potentially increasing search time. As shown in Figure 7, decreasing $|\mathsf{G}|$ indeed lowers example generation time and increases search time (other steps stay roughly the same). For RR, Comp, and Prio, the decrease in example generation time outweighs the increase in search time, and workload synthesis finishes fastest for $|\mathsf{G}| = 25$. For FQ-CoDel, however, the sweetspot is $|\mathsf{G}| = 50$. As such, one could run execute workload synthesis in parallel for different values of $|\mathsf{G}|$ and report the results from the one that finishes first. Moreover, once we have the base trace eg_0, we can potentially parallelize the generation of G into x threads, each generating $\frac{|\mathsf{G}|}{x}$ example traces. Our randomized changes from one trace to the next reduces the risk of getting duplicate traces across threads, and even if there are a few ones, it will not affect the correctness of the search algorithm.

The verification engine is efficient. Across all case studies and on average, the verification engine can verify a workload in $< 1sec$. The worst case is $\sim 10sec.$for Comp, and $< 1sec.$ in other case studies. The efficiency of the verification engine is mostly thanks to the advances in SMT solvers [6] and our efficient encoding of queues (Appendix), as well as how we encode composition. As an optimization, we can sometimes encode composition constraints such that packets leaving output queues at time t are visible in the input queues of the next module also at t (as opposed to $t+1$, §3). So, if we have a chain of L queuing modules, a packet arriving at time t is processed by all the queuing modules one-by-one but in the same time step and appears in the output at $t+1$, as opposed to $t+1+L$. When all the paths from input to output queues through the queuing modules have the same length, this optimization will not change the relative ordering of packets across queuing modules. Thus, using this optimization in Comp, the verification engine can analyze the model in 10 as opposed to 15 timesteps to get the same output.

Search optimizations are effective. Our search optimizations (§6.6) quite effective. For instance, our stochastic search algorithm is based on MCMC, which can take several rounds to converge. So, while the verification engine is reasonably fast, we try to avoid calling it if possible, e.g., when a candidate workload matches one of the bad example traces. Using this optimization, we avoided calling the verification engine in ~ 40 to 70% of the rounds, which translates to saving around 113 and 192 seconds in FQ-CoDel and Comp that are more complex. Moreover, the "look-ahead" strategy to avoid local minima is used in 5% of the rounds, helping the search algorithm to avoid local minima and plateaus.

Other observations. We found both workload minimization and broadening quite effective in making the workloads found by the search algorithm even more concise and general. They take longer for case studies like Comp that have more queues and modules, but both steps are amenable to parallelization as they shrink/broaden workloads constraints independently and one a time. So, we expect their overhead to be considerably reduced with parallelization.

8 Discussion and Future Directions

Bounded Time. We model queuing modules for a bounded number of time steps, starting with empty queues and every module in its initial state. Nevertheless, the workloads in our case studies often include "repeatable" patterns. For instance, in FQ-CoDel, Q_5 can continue causing fairness problems after 14 timesteps if it keeps sending at the rate specified in the workload. Prior work explores how to find periodic adversarial input patterns for P4 programs [32], and it would be interesting to explore similar ideas in our context. Moreover, to cover different portions of the time horizon, we can explore ways to compute a subset set of reachable states in queuing modules and start from those as oppose to the initial state. Exploring verification techniques for reasoning about unbounded time [39] is an interesting avenue for future work.

Packets vs. bits. In our prototype, metrics, and therefore, queries, and workloads, are defined in terms of packets rather than bits. We plan to extend AutoPerf to include packet sizes as extra variables, so they can be used in defining metrics and potentially reveal even more interesting workloads.

Analyzing code. As discussed in §3, we currently encode the packet processing algorithm between input and output queues of modules manually. That said, new algorithms can be easily plugged into our framework as long as they are expressible in SMT. We leave the design of high-level languages for specifying these algorithms and automatic derivation of SMT encodings from code to future work.

9 Related Work

Network Calculus. Network Calculus [15, 36] offers a uniform mathematical framework for analyzing performance guarantees. One models the input workload as an "arrival curve", which bounds the arrival pattern of bits into a network component, and the network component as a "service curve", which bounds the number of serviced bits. Using these curves and $(min,+)$ algebra, one can derive bounds on performance metrics such as throughput, latency, jitter, and loss [13, 19, 37, 38]. However, these curves need to be reasonably concise and provide tight bounds on the behavior of the network component and its input workload for the final bounds to be tight and useful. Deriving arrival and service curves is challenging, particularly for today's complex workloads, scheduling algorithms, and protocols [14].

Quantitative Reasoning. There is a line of work for reasoning about quantitative network properties: Some [31, 35, 55] extend dataplane verifiers [9, 33] to reason about quantities such as link loads and hop counts. Others [20, 23] reason about probabilistic aspects of networks (e.g., weighted ECMP) and answer probabilistic questions (e.g., the probability that packets reach a destination). These tools focus on *verification*, whereas, we focus on *synthesis* of workloads that violate performance-related properties. Moreover, these tools abstract away many low level network details as nondeterministic, which we model precisely in our queueing modules.

Synthesis in networking. Recent work explores applying synthesis to networking to generate optimized packet processing code [22, 54], synthesize network configuration [11, 12, 18], generate configuration updates [48], or synthesize control-plane repairs [7, 24]. We also use synthesis, but to generate workloads to reason about network performance.

Automated protocol analysis. Recent work uses techniques such as bounded model checking and guided search to check if congestion control algorithms can be driven into undesirable states or underutilize the network [10, 30, 53]. Khan et al. propose to train Markov models that capture the temporal behavior and throughput and delay distributions of delay-based congestion control protocols [34]. Gilad et al. use reinforcement learning (RL) to train agents that generate adversarial traces for protocols and use it to demonstrate sub-optimal performance in some RL-driven protocols [25]. We focus on generating workloads describing sets of traces in response to user-defined queries about performance problems.

Syntax-Guided synthesis. SyGuS is a general approach to program synthesis that uses a verifier together with a candidate program grammar. There have been numerous methods proposed for SyGuS, including enumerative approaches [21, 44], stochastic approaches [47, 49, 54], and logical approaches [26]. Our work is based on stochastic search [49], but is highly optimized for our queueing model.

Availability

We will release the source code for our prototype and case studies upon acceptance.

References

[1] Apache Spark. https://spark.apache.org/. Accessed: 09-2021.

[2] Mellanox Technologies - ConnectX-4 VPI. https://www.mellanox.com/files/doc-2020/pb-connectx-4-vpi-card.pdf. Accessed: 09-2021.

[3] memcached: a distributed memory object caching system. http://www.memcached.org/. Accessed: 09-2021.

[4] Mininet. http://mininet.org/. Accessed: 09-2021.

[5] NS3 Network Simulator. https://www.nsnam.org/. Accessed: 09-2021.

[6] Z3. https://github.com/Z3Prover/z3. Accessed: 09-2021.

[7] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: incrementally synthesizing policy-compliant and manageable configurations. In *CoNEXT*, 2020.

[8] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of DCTCP: stability, convergence, and fairness. *SIGMETRICS*, 2011.

[9] Carolyn Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. 2014.

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

[11] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *SIGCOMM*, 2016.

[12] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *PLDI*, 2017.

[13] C-S Chang. Stability, queue length and delay. ii. stochastic queueing networks. In *[1992] Proceedings of the 31st IEEE Conference on Decision and Control*. IEEE, 1992.

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

[15] Rene L Cruz. A calculus for network delay. parts i and ii. *IEEE Transactions on information theory*, 1991.

[16] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 2013.

[17] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI*, 2014.

[18] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *NSDI*, 2018.

[19] Victor Firoiu, J-Y Le Boudec, Don Towsley, and Zhi-Li Zhang. Theories and models for internet quality of service. *Proceedings of the IEEE*, 2002.

[20] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. Probabilistic netkat. In *European Symposium on Programming*, 2016.

[21] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: A type-theoretic interpretation. January 2016.

[22] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *SIGCOMM*, 2020.

[23] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. Bayonet: probabilistic inference for networks. 2018.

[24] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *SOSP*, 2017.

[25] Tomer Gilad, Nathan H Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. Robustifying Network Protocols with Adversarial Examples. In *HotNets*, 2019.

[26] Sumit Gulwani and Ramarathnam Venkatesan. Component based synthesis applied to bitvector circuits. Technical report, April 2009.

[27] Jiayue He, Mung Chiang, and Jennifer Rexford. TCP/IP interaction based on congestion price: Stability and optimality. In *2006 IEEE International Conference on Communications*, 2006.

[28] Toke Høeiland-Jøergensen, Paul McKenny, Dave Taht, Jim Gettys, and Eric Dumazet. The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm. RFC 8290, 2018.

[29] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. *ACM SIGCOMM Computer Communication Review*, 2013.

[30] Samuel Jero, Md Endadul Hoque, David R Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated Attack Discovery in TCP Congestion Control Using a Model-guided Approach. In *NDSS*, 2018.

[31] Garvit Juniwal, Nikolaj Bjørner, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. Quantitative Network Analysis.

[32] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. Probabilistic profiling of stateful data planes for adversarial testing. In *ASPLOS*, 2021.

[33] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[34] Muhammad Khan, Yasir Zaki, Shiva Iyer, Talal Ahamd, Thomas Pötsch, Jay Chen, Anirudh Sivaraman, and Lakshmi Subramanian. The Case for Model-Driven Interpretability of Delay-Based Congestion Control Protocols. *SIGCOMM Computer Communication Review*, 2021.

[35] Kim G Larsen, Stefan Schmid, and Bingtian Xue. Wnetkat: A weighted sdn programming and verification language. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2017.

[36] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001.

[37] Jörg Liebeherr, Almut Burchard, and Florin Ciucu. Delay bounds in communication networks with heavy-tailed and self-similar traffic. *IEEE Transactions on Information Theory*, 2012.

[38] Jorg Liebeherr, Yashar Ghiassi-Farrokhfal, and Almut Burchard. On the impact of link scheduling on end-to-end delays in large networks. *IEEE Journal on Selected Areas in Communications*, 2011.

[39] Anthony Lin. *Model Checking Infinite-State Systems: Generic and Specific Approaches*. PhD thesis, 08 2010.

[40] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *SOSP*, 2017.

[41] Steven H Low. A duality model of TCP and queue management algorithms. *IEEE/ACM Transactions On Networking*, 2003.

[42] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store. In *ATC)*, 2013.

[43] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *NSDI*, 2013.

[44] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. June 2015.

[45] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 2015.

[46] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. {SENIC}: Scalable {NIC} for end-host rate limiting. In *NSDI*, 2014.

[47] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[48] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *SIGCOMM*, 2021.

[49] Rahul Sharma and Alex Aiken. From Invariant Checking to Invariant Inference using Randomized Search. *Formal Methods in System Design*, 2016.

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

[51] Rayadurgam Srikant. *The mathematics of Internet congestion control*. Springer Science & Business Media, 2004.

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

[53] Wei Sun, Lisong Xu, Sebastian Elbaum, and Di Zhao. Model-Agnostic and Efficient Exploration of Numerical State Space of Real-World TCP Congestion Control Implementations. In *NSDI*, 2019.

[54] Qiongwen Xu, Michael D Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *SIGCOMM*, 2021.

[55] Ying Zhang, Wenfei Wu, Sujata Banerjee, Joon-Myung Kang, and Mario A Sanchez. Sla-verifier: Stateful and quantitative verification for service chaining. In *INFOCOM*, 2017.

[56] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *SIGCOMM*, 2015.

Efficient Encoding of FIFOs in SMT

A queue is specified with two parameters, its size S and the maximum number of enqueues K allowed at every time step. We define the following variables for each queue for every time step t:

1. $enqs[t][1:K]$ consists of K tuples to capture the packets that are sent to the queue at time t. These packets will enter that queue at time $t+1$,
2. $elems[t][1:S]$, consists of S tuples to captures the packets that are inside the queue at time t, and
3. an integer variable $deq\_cnt[t]$ that captures how many packets will be dequeued from this queue at time t.

We also define a set of helper boolean variables $val\_elem[t][i]$. $val\_elem[t][i]$ is true if there is a packet in $elems[t][1:S]$ and is false if $elems[t][i]$ is empty. $val\_enq[t][1:K]$ is defined similarly.

Our queues can have up to K enqueues and up to S dequeues in every time step, which makes it challenging to encode them efficiently. We first define an extra set of helper variables $tmp\_val[t][1:S]$ to denote which indexes in the queue would still have packets and which ones would become empty at $t+1$ if we were to only do $deq\_cnt(t)$ number of dequeues. Specifically, $tmp\_val[t][i]$ is true if the ith element of the queue will still contain a packet after the dequeue in that time step. To capture that, for every $1 \le i \le S$ and $1 \le d \le S$, if $i+d \le S$, we add

$$\big(deq\_cnt[t] = d\big) \rightarrow \big(tmp\_valid[t][i] = val\_elem[t][i+d]\big)$$

Otherwise, we add

$$\big(deq\_cnt[t] = d\big) \rightarrow \big(\neg tmp\_val[t][i]\big).$$

We have some standard constraints to shift the packets in $elem$ forward depending on the $deq\_cnt(t)$.

The next set of constraints handle the enqueues in a "sliding window" fashion. That is, starting from the head of the queue, we consider all possible $K+1$ consecutive positions in the queue to find a window where the first element has a packet and the next K are empty. This will be the tail of the queue and where we will be enqueuing the new packets. That is, for every $1 \le i \le S-K$ and $1 \le j \le K$, we add the following constraint:

$$tmp\_val[t][i] \wedge \neg tmp\_val[t][i+1] \rightarrow elem[t][i+j] = enqs[t-1][j]$$

We add extra constraints for the start and end of the queue and when there is not enough space for K packets.

Finally, we add constraints to make sure there is no "hole" in the queue. That is, the queue starts with a sequence of packets, and when there is an empty space, the rest of the queue will be empty as well. Specifically, for every $1 \le i < S$, we add:

$$val\_elem[t][i] \vee \neg val\_elem[t][t]$$

Definition of $\mathsf{cost_S}$

Consider a workload $\mathsf{wl} = \wedge_{i=1}^{k} \mathsf{con}_i$, where $\mathsf{con}_i = \forall t \in [T_{1_i}, T_{2_i}] : \mathsf{lhs}_i \oplus_i \mathsf{rhs}_i$. $\mathsf{cost_S}(\mathsf{wl})$ is defined in the following way:

$$\mathsf{cost_S}(\mathsf{wl}) = \Sigma_{\mathsf{i=1}}^{\mathsf{k}} \mathsf{queue\_cnt}(\mathsf{spec_i}) + \mathsf{interval\_cnt}(\mathsf{wl})$$

where $\mathsf{queue\_cnt}(\mathsf{spec_i})$ is the number of queues constrained by con_i, which is equal to the number of queues specified in lhs_i. $\mathsf{interval\_cnt}(\mathsf{wl})$ captures the degree of time fragmentation in the workload and is defined as the number of non-overlapping time intervals with unique sets of constraints. As an example, suppose wl has three constraints, $\forall t \in [1,15] : cenq(Q1,t) \ge 2$, $\forall t \in [3,7] : cenq(Q2,t) \le 5$, and $\forall t \in [5,10] : aipg(Q5,t) = 3$. These three constraints are specifying a traffic pattern over five non-overlapping time intervals ($[1,2],[3,4],[5,7],[8,10],[11,15]$) each with a unique set of constraints. $\mathsf{interval\_cnt}$ is equal to five in this example. We favor workloads that cause less time fragmentation as they are less likely to overfit to the example sets and be more interpretable.