Formal Methods for Network Performance Analysis

Mina Tahmasbi Arashloo
University of Waterloo

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 continuously evolving 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.

We explore a different approach: using formal methods to analyze 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. 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 packet scheduling algorithms and a small leaf-spine network and generating concise workloads that can cause throughput, fairness, starvation, and latency problems.

1 Introduction

New network functionality is often analyzed, both empirically and analytically, to predict how it would perform under different kinds of input traffic. However, it is growing increasingly difficult to do such analysis in a manner that is reasonably *exhaustive*, i.e., does not miss traffic patterns that are probable to occur in production, and *general*, i.e., is not only applicable to a limited set of network functionality.

To see why, consider the existing empirical approaches, i.e., using simulators, emulators, and testbeds [1–3]. These approaches allow operators to realize a model of their network in software and/or hardware, push a concrete sequences of packets through it, and measure how well the network performs for that specific input traffic pattern. As such, operators have to pick and choose which input traffic patterns to try. This leaves room for unexplored traffic patterns that may experience poor performance because of overlooked corner cases and bugs. The problem is only exacerbated as new networked applications emerge, expanding the set of traffic patterns that a network may face in production.

On the analytical side, there is a substantial body of work that applies techniques from queuing theory and network calculus to derive bounds on performance metrics such as throughput, delay, and loss [4–9]. However, to obtain tight and useful bounds, the network functionality and the arrival pattern of traffic need to be closely approximated with concise and well-behaved mathematical functions. This limits the set of network functionality and traffic patterns that can be reasoned about using these frameworks [10], a problem that is, again, aggravated with the continuous evolution of networked applications and the network functionality that supports them.

This paper explores an alternative approach: *using formal methods to analyze network performance*. That is, we would like to use logical formulas to model packets, how packets are processed by each piece of network functionality as they traverse the network, and how performance metrics such as throughput and delay change over time. We can then use verification and synthesis techniques from the formal methods community to exhaustively explore the space of all possible traffic patterns and find those for which the network cannot provide satisfactory performance.

Why use formal methods? Because they can nicely complement existing empirical and analytical approaches for performance analysis. Unlike empirical approaches, they do not need to explicitly try out every single traffic pattern to find ones that experience performance problems. Moreover, they enable us to reason about network functionalities that are expressible in logical formulas, many of which may be not feasible to approximate in a way that is suitable for existing analytical approaches. Finally, past efforts in using formal methods to solve networking problems have proven quite successful. Over the past decade, researchers and practitioners in both academia and industry have coupled advances in formal methods tools and techniques with domain-specific optimizations to rigorously reason about the functional correctness of various aspects of networks [11–28]. This provides further encouragement that formal methods could bring similar benefits to reasoning about *performance*.

Realizing this vision is not without its own unique challenges. First, performance metrics (e.g., throughput and delay) 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. As such, reasoning about performance requires reasoning about the enormous space of possible packet-level interactions and finding those that can lead to unsatisfactory performance. For that, we have found efficient ways to encode interactions that can significantly affect performance, e.g., interactions across network queues, in Satisfiability Modulo Theories (SMT) formulas and over a bounded number of time steps.

Second, when it comes to performance analysis, finding a single counter-example is not always useful. In this case, a counter-example is an assignment to the model's logical variables that leads to poor performance, where the model variables describe packets and the order and time at which they enter the network. Such a detailed packet trace is not easily interpretable. Moreover, unlike counter-examples for correctness properties, it may not even point to a bug, but an extremely rare scenario where providing sub-optimal performance does not have tangible consequences.

We argue that a more useful output (also a more challenging one to produce) is a traffic pattern or *workload* that can succinctly capture the commonality between a whole *set* of packet traces that can experience poor performance. To generate such workloads efficiently, our main insight is to use syntax-guided synthesis (SyGuS) [29]. That is, we define a language for specifying workloads. Then, we systematically search through the space of all workloads to find one such that *all* packet traces represented by it experience poor performance. The search algorithm, inspired by prior work on invariant synthesis [30], is based on Markov Chain Monte Carlo (MCMC), 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.

To demonstrate the effectiveness of our approach, we apply it to packet scheduling algorithms as well as a small leaf-spine network and ask queries about throughput, fairness, latency, 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.

These results provide an encouraging indication that using formal methods to analyze network performance has the potential to grow into a valuable tool for understanding network behavior and making networks more robust. Our proof-of-concept prototype can analyze compositions of a few tens of network components, modeling small-scale networks or host-based scenarios where flows from different applications or VMs contend for end-host resources across a few layers of classifiers and packet schedulers. That said, we recognize that, similar to data and control-plane verification tools that have matured over a decade to scale to large-scale networks [31], much work needs to be done to improve the scalability of formal methods tools for network performance analysis. We discuss potential future research directions in §9.

2 Overview and Motivation

In this section, we use a buggy packet scheduler to demonstrate our approach in using formal methods to reason about performance.

The scheduler. FQ-CoDel [32] is the default queuing discipline in Linux. It is a hybrid packet scheduler and active queue management (AQM) algorithm. Flows are classified into queues which are managed by the CoDel AQM algorithm. The scheduler decides which queue gets to transmit next. It prioritizes the transmission of the first few packets of new flows so that short latency-sensitive flows with a few packets are not blocked by longer flows.

Our motivating example is inspired by the scheduler in FQ-CoDel. When a packet comes in, it is first classified (e.g., based on its 5-tuple) and assigned to a queue. For simplicity, let's assume hash collisions are rare and each queue is holding packets of one flow at any point in time. The scheduler maintains a list of pointers to queues with potential short flows called *new_queues*, and another list called *old_queues* with pointers to all the other queues with outstanding packets. At a high level, queues in *new_queues* are prioritized over those in *old_queues*, and the queues in the same list are serviced using a deficit-round-robin (DRR)-like algorithm [33].

How does the scheduler determine which list each queue belongs to? Suppose the incoming packet is assigned to q_i. If q_i is in neither of the lists, it means it has not received packets recently and this could be the start of a short flow. So, the scheduler will add a pointer to q_i to the end of *new_queues*. Otherwise, the queue will remain in its current list. In both cases, the packet is enqueued in q_i.

On dequeue, the scheduler first looks at *new_queues*. Suppose q_h is the queue at the head of the list. It will be selected to send a packet unless (1) it is empty, in which case it is removed from the $new_{q}ueues$ and marked as inactive, or (2) it has packets but has already sent at least a (configurable) quantum of bytes, in which case it is not considered a short flow, is removed from *new_queues* and is inserted at the end of *old_queues*. If q_h is not eligible, the scheduler moves on to the next queue in *new_queues*.

The bug. When a queue in *new_queues* is empty, it is immediately deactivated. When it receives another packet, it is placed in *new_queues* and gets priority over the queues in *old_queues*. This can potentially cause starvation for queue in *old_queues*. When proposing the separation between new and old queues, the FQ-CoDel RFC [32] warns against this bug: "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,

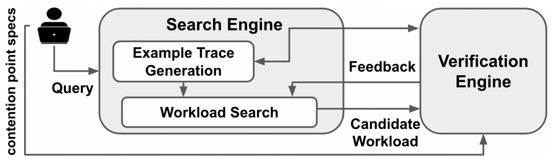


Figure 1: Overview of workload synthesis.

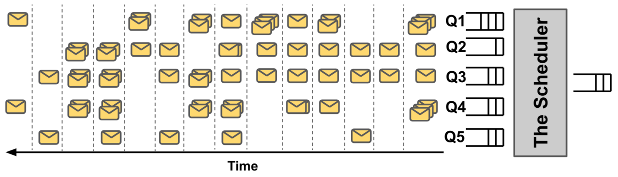


Figure 2: An example trace.

even with a small number of active flows, if the flow providing packets to the queue in question transmits at just the right rate." To avoid this problem, the RFC suggests that when a queue in *new_queues* becomes empty, it should be first demoted to the *old_queues*, and only deactivated if it still stays empty after all the old queues are visited on subsequent dequeues.

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 traffic patterns 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 arrival patterns 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 our approach using formal methods and workload synthesis to reason about the performance of network components like our example scheduler.

Modeling contention points (§3). First, the users specify which network component(s) they are interested in, and if there are more than one, how those components are connected together. Given this input, the verification engine generates logical variables and constraints that model the network components and their interactions. Each component is modeled as one or more *queuing modules*, each with n input queues, m output queues, and a processing block in the middle. The processing block takes packets from input queues, processes them, and puts the resulting packets into output queues.

Here, we model the scheduler as a queuing module with five input queues and one output queue (Figure 2). Specifically, in the verification engine, we generate SMT formulas 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. Moreover, the processing block selects one input queue according to the scheduler's dequeue logic, dequeues a packet from it, and places the packet into the scheduler's output queue. The formulas describe how the scheduler state, e.g., *new_queues* and *old_queues* lists, and queue contents at time t connects with those at time $t+1$.

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

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

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., $\lfloor T/5 \rfloor$)?". Note that we use lower case letters for variables and upper case letters for literals and constants.

Introducing workloads (§5). At this point, we can ask the verification engine to use an SMT solver like Z3 [34] to find an input packet *trace* that satisfies the query. A packet 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 scenario in which Q_5 receives more than its fair share of the bandwidth, it is not easy to interpret as it specifies the ordering and timing of the entry of every single packet: 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? Even if these details actually do matter, it is not clear if the trace is actually pointing to a subtle but prominent performance problem, or just an extremely rare and uninteresting scenario.

Rather than output a single packet trace, our search engine synthesizes a *workload* that can concisely describe a set of packet traces that can cause performance problems:

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

$cenq(q,t)$ is the total number of packets that enter q by $t+1$, and $aipg(q,t)$ is the inter-packet gap between the last two packets entering q by $t+1$. 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 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, so 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. This ensures Q_5 becomes empty and gets deactivated after dequeuing each packet (the bug). So, it is activated as a new queue when it receives its 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 ensures that the other queues are always backlogged, which comes from the assumption specified in the query.

Synthesizing workloads (§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 [30, 35], 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 if 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 is found and returned to the user, who can either terminate the analysis or ask for other workloads that satisfy the query.

User Interface. Similar to many other existing work that use formal methods for networks, our queries and workloads are specified as logical expressions. Moreover, the encoding of packet processing algorithms and protocols into logic is done manually. To enable widespread adaptation of formal approaches, it is important to develop front ends that interface with these "logical backends", abstract away the details of logical expressions and formulas, and enable users to interact with them using higher-level and more familiar interfaces. In fact, there are several ongoing efforts for providing higher level query interfaces and automated generation of logical models and SMT formulas from implementations [36–40]. With the right interface in the middle, the front-end and the logical back-end can evolve independently. As such, for our case study in §8, we create a simple front-end to demonstrate an example of one such interface, and leave the design of a more general front-end for future work.

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 that describes how to deliver data from one layer to the next. From source to destination, packets go from socket buffers, to queues in packet schedulers in the end-host stack (e.g., Linux qdiscs), to NIC transmit queues on the sender. Then, they traverse switch input and output queues in the network, and 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 it traverses and the frequency at which those queues are selected to pass on their traffic. As such, queues are a fundamental part of our model.

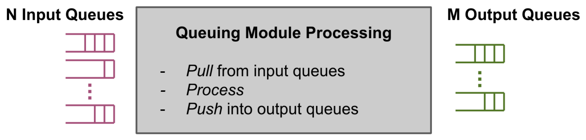


Figure 3: A queueing module.

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. Every timestep, the processing block takes a batch of packets from the input queues, processes them, and puts the resulting packets 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).

Composition. Queuing modules can be easily composed by feeding the output queues of one module to the input queues of another (e.g., figures 6 and 9). So, one could start by modeling a single bottleneck, e.g., a qdisc, or a NIC/switch scheduler, and compose them together to reason about segments or paths in the network (§7 and §8). We can even 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 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. Modeling time based on dequeues is a natural choice. It is coarser-grained than wall-clock time, and since our verification engine performs bounded model checking over time, this helps performance problems manifest over fewer time steps. It is also fine-grained enough for capturing the arrival order of packets within and across time steps. To capture the arrival order within a time step, we put an upper bound K on the number of packets that can enter a queue between dequeues. This bound allows us to create K variables $p_{t_1}, \cdots, p_{t_K}$ to capture the order at which those K packets enter the queue at time t (p_{t_i} comes before p_{t_j} if $i < j$). It also helps us avoid finding "trivial" workloads that simply flood the queuing module every time step with an unrealistically large number of packets.

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, (2) $elems[t][1:S]$, consists of S tuples to capture 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 Appendix A.

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 ($A.out_1$) is connected to the first input queue of module B ($B.in_1$). 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 $B.in_1$. We provide two different kinds of composition. In *sequential* composition, $B.in_1.enqs[t][1:k] = A.out_1.elems[t][1:k]$, that is, packets from $A.out_1$ will appear in $B.in_1$ at time $t+1$. In contrast, in *immediate* composition, packets leaving $A.out_1$ at time t will be visible in $B.in_1$ in the same time step. We discuss examples of different kinds of composition and their implications in our case studies in §7 and §8.

Modeling packet-processing algorithms. The packet-processing algorithm in a queuing module sets 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. It also decides which one of those packets to move to the $enqs[t]$ of which of the output queues, potentially changing, dropping, or cloning some packets in the process. As long as an algorithm's logic can be expressed in SMT, we can plug it into our framework.

4 Performance Queries

Performance queries are logical formulas over one or more performance metric. Users can define their metrics of interest to be tracked for all or a subset of queues in the queuing modules. They can then ask queries about the value of a certain metric for one queue or a set of queues, or compare the values of metrics between different 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, metric d that tracks the maximum delay experienced

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

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

by packets in a queue, can be defined in the following way:

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

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 values of metrics for one queue or a set of queues, or compare the values of metrics for two 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, users can specify base_wl, a workload (formally defined in Figure 4 and §5) that constrains the space of traces the user is interested in. That is, the final workload returned 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. For that, we can use the following query:

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

where $cenq(q,t)$ and $cdeq(q,t)$ track the total number of packets that have entered and exited q by end of t.

Similarly, we can investigate fairness between two queues by assuming minimum input rates for both in base_wl and comparing the number of packets they transmit:

$$\begin{aligned}
\mathsf{base\_wl} = {} & \forall t \in [1,T] : cenq(Q_1,t) \geq R_1 \cdot t \\
& \land \forall t \in [1,T] : cenq(Q_2,t) \geq R_2 \cdot t \\
\mathsf{base\_wl} \rightarrow {} & \exists t \in [T,T] : cdeq(Q_1,t) - cdeq(Q_2,t) \geq T/2
\end{aligned}$$

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

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. This can help users 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

Algorithm 1: Workload synthesis search.

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

$\forall t \in [T_1, T_2] : \mathsf{lhs} \oplus \mathsf{rhs}$, where T_1 and T_2 are integers denoting the interval of time over which con constrains the input traffic, lhs is either a metric for one queue, or the aggregate of a metric over a set of queues, and $\oplus$ is a comparison operator $(>, \geq, <, \leq, =)$ that shows how the lhs will be constrained by the rhs, which is 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, as a result, does not leave the queue idle, is part of this workload. It does not matter if the traffic enters 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, 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 are 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 workloads in a concise and intuitive manner. We define our workloads over two metrics: (1) cumulative enqueues ($cenq(q, t)$), the total number of packets that enter q by the end of time step t, and (2) arrival inter-packet gap ($aipg(q, t)$), the inter-packet gap between the last two packets that enter q by time t.

While small, this is a quite expressive set. *cenq* constrains the total number of packets entering the queue, independent of the exact time they arrive. So, it can abstract away the timing details of traces when they are not important for the query. *aipg*, on the other hand, constrains the gap between packets and their arrival pace. So, it can capture low-level timing details if necessary in answering the query. Together, they create a good balance in capturing the commonalities of traces that satisfy a query, abstracting away unnecessary details and including necessary ones.

Our experience in the case studies has shown that this set is capable of expressing a variety of workloads. But, we view this as a suitable starting point and not necessarily the final answer. We hope that as using formal methods, and specifically 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, if needed, any metric that can be encoded in SMT can be easily added to our search algorithm (see §8 for examples).

6 Synthesizing Answers

The search engine uses a guided randomized search over the space of workloads to find one that satisfies the query. The search algorithm (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 [20, 30, 35]. 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 the workload, i.e., check if all the traces in the workload satisfy the query (line 6). If yes, 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 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 [34] to check if the following formula is satisfiable

$$\mathsf{model} \land \mathsf{base\_wl} \land \mathsf{wl} \land \neg \mathsf{qry}$$

where model is the logical encoding of the queueing modules the user is interested in (§3).

If the formula is satisfiable, there is at least one trace that is (1) valid, i.e., satisfies the constraints specified in model such as the maximum number of packets that are allowed to enter a queue between dequeues, (2) satisfies both base_wl (the space of traces in which user is interested (§4)) and wl, and (3) does not satisfy the query. This means that our current candidate workload, wl, is not a suitable answer to the query. So, this trace is returned to the search algorithm to guide the synthesis of the next candidate workload.

If the formula is not satisfiable, either (1) base_wl or wl or their combination with respect to model is *infeasible*, meaning that no valid trace can satisfy their constraints and they actually represent the empty set, or (2) there are no valid traces in $\mathsf{base\_wl} \land \mathsf{wl}$ that do not satisfy the query. Only in the second case wl is a valid and non-trivial answer to the query.

To distinguish between these two cases, the search engine asks the verification engine if $\mathsf{model} \land \mathsf{base\_wl}$ is satisfiable (line 2). If not, the set of valid traces specified by base_wl is 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 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, the search algorithm synthesizes another workload to try. The next candidate workload, $\mathsf{next\_wl}$, is a mutation of the previous one, wl. 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 [30, 35].

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, $\mathsf{next\_wl}$ becomes the current candidate (line 11). If $\mathsf{next\_wl}$ has a higher cost, to avoid getting stuck in local minima, the algorithm may still choose to make the transition with a probability proportional to the difference in $\mathsf{next\_wl}$ and wl's costs (line 12). The algorithm repeats this process until it finds the next candidate workload.

6.3 The Cost Function

Intuitively, a good cost function should (1) favor workloads when they include 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 [30], we quantify these criteria into a cost function using *example traces*.

We create two sets of 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 that 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).

In our experience, $\mathsf{cost_E}$ can effectively guide the search toward workloads that satisfy the query. But there could be multiple such workloads. So, we define another function $\mathsf{cost_S}(\mathsf{wl})$ that favors *concise* workloads, i.e., those with fewer constraints that constrain fewer queues over longer, less fragmented periods of time (details in Appendix B). We define our final cost function as $\mathsf{cost}(\mathsf{wl}) = \mathsf{C_E} \cdot \mathsf{cost_E}(\mathsf{wl}) + \mathsf{C_S} \cdot \mathsf{cost_S}(\mathsf{wl})$. 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 satisfies 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 it towards a more concise answer.

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.

6.4 Generating The Example Sets

Before the search starts, the search engine asks the verification engine to generate traces for G and B. A trace eg is 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 $\neg(\vee_{j=1}^{i-1} \mathsf{eg_i} \neq \mathsf{eg}_j)$ ensures that the trace 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. For that, 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 $\mathsf{local\_mods}_i = \wedge_{j=0}^{P} \mathsf{eg}_i[q_j][t_j] = v_j$. This way, 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 $\mathsf{match}(\mathsf{wl}, \mathsf{B})$ 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 $\mathsf{match}(\mathsf{wl}, \mathsf{G})$, the algorithm may repeatedly be directed towards a part of the search space where there are no suitable answers.

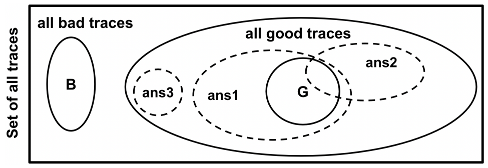


Figure 5: Relationship between answer workloads and G and B.

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, so that the workload found by the search algorithm is sufficiently general. To do so, the search engine asks the verification engine to create a *base* trace that it will use 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. This helps keep the search focused. To see why, suppose we find an eg_0 that satisfies the query and in which only Q_1, Q_2, and Q_3 have traffic. When generating the rest of G, we add constraints that make sure the queues that have no traffic in eg_0 stay empty in the rest of the traces as well. So if there is another set of traces with traffic in, say, Q_2, Q_4, and Q_5, that satisfy the query, they will not be included in G, making sure G's traces are not radically different from each other.

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", i.e., 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 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 and only allows more than one packet per time step in the base trace if 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 previous traces, and is different from the previous trace in at least P random places. There are two differences: for the ith trace eg_i, we add extra constraints so that queues that have no traffic in eg_0 (see optimization criteria 1) have no traffic in eg_i either, and we 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 Optimizations

We have employed several optimizations to improve the synthesis process and the final answer. We discuss some of the major ones here and the rest in Appendix C.

Reducing the search space. If a queue q has no traffic in our base trace eg_0 (§6.4), it means that as long as the other "non-idle" queues have traffic, its traffic is either not important or has to be 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. These constraints will be removed in post-processing if not necessary.

Post-processing. Once the search engine finds a workload wl that satisfies the query, it creates a new workload ans that includes all the constraints from wl and base_wl. It then performs "workload shrinking" (Algorithm 1, line 13) by removing the constraints in ans one at a time and checking if ans still satisfies the query. This helps remove any constraint that is added to base_wl during example generation or to wl during search but is not necessary for satisfying the query. Next, we try "workload broadening". For a queue Q_i that is not in ans and a constraint con in ans, if con's left hand side is $m(Q_j,t)$, it is changed to $\Sigma_{q\in\{Q_i,Q_j\}} m(q,t)$, and if it is $\Sigma_{q\in S} m(q,t)$, it is changed to $\Sigma_{q\in S\cup\{Q_i\}} m(q,t)$. If the workload still satisfies the query, this helps include even more traces in the workload.

Reducing calls to the verification engine. Each call to the verification engine can be expensive as it checks the satisfiability of non-trivial formulas. So, if the search algorithm selects a candidate workload that matches a trace in B, it moves 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.

Escaping local minima. 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 the next candidate by applying a sequence of moves in §6.2 to generate the next workload. If it still cannot make enough progress in another several rounds, it restarts the search from a workload with no constraints.

7 Case Study: Packet Scheduling

We have prototyped our techniques in a tool we call FPerf in ~10K lines of C++ code, which is publicly available [41]. We use Z3 [34] in the verification engine for checking the satisfiability of SMT formulas. In this section, we describe our experience in using FPerf to analyze packet scheduling algorithms. Our goal is to explore *expressiveness*, i.e., whether our workload language can express a wide range of workloads in answering queries, *interpretability*, i.e., whether the final workloads are concise, intuitive, and interpretable, and *tractability*, i.e., whether workloads are generated in reasonable human timescales (i.e., minutes).

7.1 Stand-alone Schedulers

The priority scheduler is a single queuing module with four input queues 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 a packet from it, and puts the packet in the output queue. In a strict priority scheduler, lower priority queues may get starved, which we quantify with the metric $blocked(q,t)$ defined as the number of consecutive time steps that q has packets but is not chosen for transmission. We then ask $\exists t \in [1,T] : blocked(Q_3,t) > 5$.

Starting from $T = 5$, we increment T until the verification engine finds a satisfying trace in the example generation phase at $T = 7$. Then, the search algorithm finds $\forall t \in [1,7] : \Sigma_{q\in\{Q_1,Q_2\}} cenq(q,t) \geq t \wedge \forall t \in [1,7] : cenq(Q_3,t) \geq 1$. That is, to satisfy the query, Q_1 and Q_2, which have higher priorities than Q_3, should collectively receive a consistent flow of traffic (constraint 1), and Q_3 should at least have one packet to be considered blocked (constraint 2). While the answer is not surprising, it demonstrates FPerf's ability to abstract away the details of which higher priority queue receives packets at exactly what time step, only capturing the necessary details.

The 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 §9). 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 Q_2 with query $\forall t \in [10,10] : cdeq(Q_3,t) - cdeq(Q_2,t) \geq 3$ and base workload $\wedge_{i=1}^{5} \forall t \in [1,10] : cenq(Q_i,t) \geq t$, the verification engine cannot find any traces that satisfy the query.

Now suppose we relax base_wl to restrict the average rate of traffic every 5 time steps as opposed to every time step, i.e., have the queues receive at least 4 packets every 5 time steps: base_wl $= \big(\wedge_{i=1}^{5} \forall t \in [5,5] : cenq(Q_i,t) \geq 4\big) \wedge \big(\wedge_{i=1}^{5} \forall t \in [10,10] : cenq(Q_i,t) \geq 8\big)$. FPerf finds $\forall t \in [1,4] : \Sigma_{q\in\{Q_1,Q_2,Q_4,Q_5\}} cenq(q,t) \leq 0 \wedge \forall t \in [1,4] : cenq(Q_3,t) \geq t \wedge$ base_wl, which describes a workload where all queues except Q_3 receive no packets in the first four time steps and receive a burst of at least 4 packets at time 5, while Q_3 continuously receives traffic. So, 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.

FQ-Codel. This case study analyzes the buggy scheduler inspired from the FQ-Codel qdisc [32]. 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 runs.

7.2 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 [42–44]. 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 [44] demonstrates this using an example, which we analyze in this case study.

Suppose two tenants reside on a server with multiple CPU cores. Tenant 1 runs spark [45], and tenant 2 runs both spark and memcached [46]. All applications are multi-threaded and can use all the cores. We want to ensure that the two tenants fairly share the network bandwidth, and tenant 2's memcached traffic is prioritized over its own spark traffic. One option, described in [44], 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 enforce policies per core not across cores.

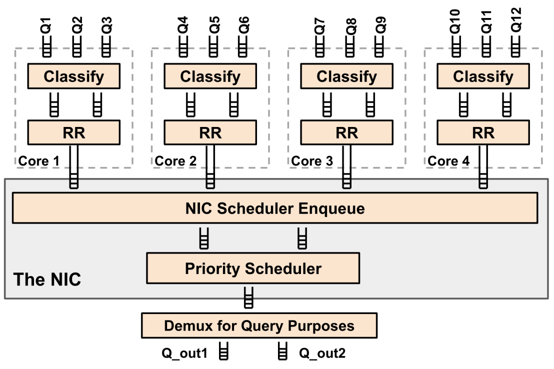


Figure 6: Setup for the case study in §7.2.

Figure 6 shows how we model this in FPerf 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 each core, a queuing module first classifies traffic from that core's input queues into two output queues, one for each tenant. Then, a round-robin scheduler shares bandwidth between the two tenants into the NIC's transmit queue. On the NIC, a module classifies traffic from the cores into two output queues, one for spark and one for memcached traffic. Finally, a priority scheduler that prioritizes memcached traffic decides what packet to send out of the NIC. We add a "dummy" module that takes the output from the NIC and "demultiplexes" it into Q_{out1} queue for tenant 1 and Q_{out2} for tenant 2 to 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. So, we ask whether tenant 1 and tenant 2 will get equal access to the NIC output link:

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

$S_{tenant1}$ 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 $S_{tenant2}$. That is, in base_wl, we ensure that both tenants receive a steady stream of packets. For this query, FPerf finds the workload

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

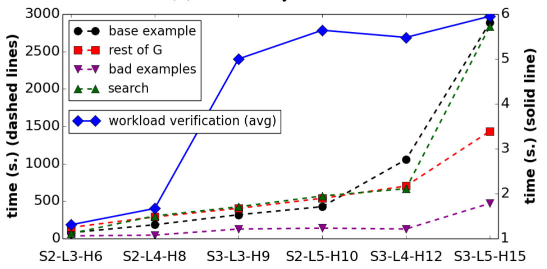
where S_{rest} is $\{1,\cdots,3,5,\cdots,8,10,\cdots 12\}$. That is, if tenant 2's memcached runs on core 3 and tenant 1's spark on core 2, they

Phases		**Prio**	**RR**	**FQ-C**	**Comp**	**LS-T**	**LS-L**
Example Generation	Generating base trace (s)	0.3	1.0	3.1	70.3	59.5	71.5
	Generating good set (G) (s)	6.8	309	346	321	310	632
	Generating bad set (B) (s)	1.2	3.0	7.1	179	74.1	69.6
Verification Engine	Verifying workload (avg) (s)	0.02	0.02	0.11	0.65	0.73	0.94
	Verifying workload (max) (s)	0.04	0.13	0.86	7.60	3.26	2.18
	Verifying query (avg) (s)	0.03	0.04	0.10	0.81	1.66	1.46
	Verifying query (max) (s)	0.06	0.18	0.73	9.90	4.09	2.26
Search (see §6.5)	# Rounds	65	268	769	361	949	117
	# Rounds w.o/verification	39	107	537	131	836	65
	# Rounds w/"look-ahead"	1	11	44	20	162	1
	Total search time (s)	2.4	59	223	461	420	121
Post Processing	Workload shrinking (s)	0.2	0.0	0.9	107	25.9	16.4
	Workload broadening (s)	1.0	0.0	0.1	45	0.0	2.6
Total Time (min.)		0.2	6.2	9.6	18.5	13.8	14.0

(a) Statistics from running FPerf 10 times for each case study. **Parameters:** 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 $0.03 \cdot max(\mathsf{cost_E})$, $\frac{\mathsf{C_E}}{\mathsf{C_S}} = 10$, and number of rounds of slow progress tolerated before look-ahead and restart is 10 and 20 respectively.

	Prio	**RR**	**FQ-C**	**Comp**	**LS-T**	**LS-L**
Queuing Modules	1	1	1	11	23	23
Queues	5	6	7	29	66	66
Boolean Vars ($\times 1000$)	0.8	0.2	2.6	10.6	21.3	21.3
Integer Vars ($\times 1000$)	0.6	1.1	1.9	7.3	23.2	23.2
Constraints ($\times 1000$)	7.2	13	2.2	93.8	45.8	45.5
Max timesteps	7	10	14	10	10	10

(b) Case study statistics



(c) Workload synthesis statistics for the latency query on leaf-spine networks of increasing size. S_i-L_j-H_k has i spines, j leaves, and i hosts per leaf to avoid oversubscription ($k = i \times j$). Parameters are same as figure 7a except $|\mathsf{G}| = |\mathsf{B}| = 25$ (see §9).

Figure 7: Case study statistics and results (§7 and §8).

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:

$$\forall t \in [1,10] : cenq(Q_8,t) \geq t \wedge \forall t \in [1,10] : \ cenq(Q_{11},t) \geq t$$
$$\wedge \forall t \in [1,10] : \ \Sigma_{q\in\{Q_2,Q_5\}} cenq(q,t) \geq t$$
$$\wedge \forall t \in [1,10] : \ cenq(Q_{10},t) \geq t$$
$$\wedge (\wedge_{i\in rest} \forall t \in [1,10] : \ cenq(Q_i,t) \leq 0).$$

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 [44].

7.3 Tractability

Tables 7a and 7b summarize statistic about different phases involved in workload synthesis for the packet scheduling case studies across 10 runs. We use a server with 4-socket NUMA-enabled Intel Xeon Gold 6128 3.4GHz CPU with 6 cores per socket. For Comp, we only include the results for the second query as, compared to the first query, its analysis is more involved in all synthesis phases.

Generating G is expensive. This is not surprising: when generating trace eg_i, we constrain it to have different randomly chosen values from eg_{i-1} in $p = P$ random 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 (§6.4). The more such calls, either due to the sheer size of G, or because we have to retry a lot for each trace, the longer generating G will take.

Compared to Prio and Comp, it takes more tries to find a trace for RR and FQ-Codel (average of 16 and 6 tries, respectively). This is because the answers to their queries are more sensitive to the timing of packets in the trace: RR needs a specific kind of burstiness and FQ-CoDel needs a specific rate. So, 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, translating to a maximum of 20 tries per example trace. Instead, we can potentially set P to the moving average of the p that has worked for the previous traces and reduce the number of retries, and consequently, the total time for generating G.

Another option is to reduce $|\mathsf{G}|$, potentially increasing search time as shown in Figure 10 (Appendix). For all but FQ-Codel, the decrease in example generation time outweighs the increase in search time, and workload synthesis is fastest for $|\mathsf{G}| = 25$. For FQ-CoDel the sweetspot is $|\mathsf{G}| = 50$. One could execute workload synthesis in parallel for different values of $|\mathsf{G}|$ and use the results from whichever finishes first. Moreover, once we have the base trace eg_0, we can potentially parallelize the generation of G into x threads, each generating $\frac{|\mathsf{G}|}{x}$ 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, it will not affect the correctness of the search.

The verification engine is efficient. The verification engine can verify a workload in $< 1sec$ on average. The worst case is $\sim 10 sec.$ for Comp, and $< 1 sec.$ in other case studies. The efficiency of the verification engine is thanks to the advances in SMT solvers [34], our efficient encoding of queues (Appendix), and our choice to use immediate composition.

In *immediate* composition, packets leaving output queues at time t are visible in the input queues of the next module at time t, as opposed to $t+1$ in sequential composition (§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 final output queue at $t+1$, as opposed to $t+1+L$. In Comp, all the paths from input to output queues through the queuing modules have the same length. So, if we use immediate composition across *all* modules, it will not change the relative ordering of packets across queuing modules and allows us to analyze the model in 10 as opposed to 15 time steps and get the same output.

Optimizations (§6.5) are effective. For instance, when a candidate workload matches one of the bad example traces, we can move on to the next candidate without calling the verification engine. Using this optimization, we avoided calling the verification engine in ~ 40 to 70% of the rounds, which translates to saving ~ 113 and 192 seconds in FQ-CoDel and Comp that are more complex. Moreover, The "look-ahead" strategy is used in 5% of the rounds, helping the search algorithm to avoid local minima and plateaus.

8 Case Study: A Small Leaf-Spine Network

In this section, we use FPerf to analyze a small leaf-spine network (Figure 8). Our goal is to demonstrate the expressiveness of our model and the generality of our techniques.

Modeling switches. We model our switches after input-queued switches with virtual output queues (VOQs). Suppose the switch has P ports. Each input port i has P VOQs, where the jth VOQ stores packets that are destined for output port j. The switch crossbar decides which input ports can simultaneously send packets to which output ports without interfering with each other, and delivers packets from the corresponding VOQs at those input ports to their destination output ports.

Figure 9 shows how we model this in FPerf. A switch with P ports is a composition of $P+1$ queuing modules. There are P forwarding modules, one for each input port. The ith forwarding module takes a packet from input port i, decides the destination port j it should be forwarded to, and places the packet in the jth VOQ for port i. The crossbar queuing module models the switch crossbar. In each time step, using the iSlip algorithm [47], it matches input ports and output ports such that each input port is matched with at most one output port and vice versa. If an input port i is matched with an output port j, the crossbar moves a packet from the jth VOQ at port i to the output queue for port j. iSlip and its variants are widely used in switching fabrics as they provide high throughput in the crossbar and fairness across inputs.

Packet metadata and workload metrics. The per-packet metadata variables include dst, a variable representing the final destination of the packet, and $ecmp$, a variable with values in $[0, S]$ where S is the number of spines switches, representing the result of the ECMP hash of the packet's flow id modulo the number of spines. We extend example traces and our algorithms for generating them (§6.4) to include packet metadata (details are in Appendix D.1). Moreover, we add two corresponding metrics, $dst(q,t)$ and $ecmp(q,t)$, to our workload language to track the values of these variables for packets entering q at time t. These metrics, together with $cenq(q,t)$ and $aipg(q,t)$, can be used in the synthesized workloads to describe a range of traffic patterns and flows.

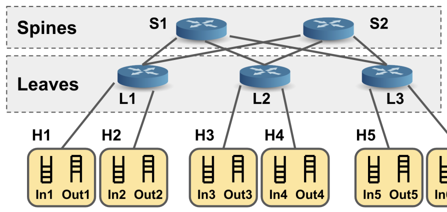


Figure 8: A small leaf-spine network as case study (§8).

User Interface. To create a model of a network of switches, the users need to specify the topology (switches and links) and the forwarding rules (mapping per-packet metadata to an output port) for each switch. For a leaf-spine network, FPerf provides a special interface that only requires specifying the number of leaves and spines. For the base workload, the users can provide a list of constraints using an interface that abstracts away the logical operators and expressions in Figure 4. Each constraint is either (1) [t1, t2] q.m $\oplus$ c, constraining metric m for queue q against a constant ($\oplus$ is any comparison operator), (2) [t1, t2] (q1.m + ... + qn.m) $\oplus$ c, (3) [t1, t2] q1.m $\oplus$ q2.m, or (4) [t1, t2] (q1.m + ... + qn.m)/t $\oplus$ c, constraining a metric's value for one or more queues over time. Queues are identified by switch id and port number.

For queries, the users provide a list of questions with a similar interface, asking if the value of a metric over a time period for one or more queues can go above or below a threshold. There are two special shorthands for common queries: q.avg_rate is the average input rate for queue q, which translates to q.cenq/t, and lat(s1, ..., sn) is the latency through the specified sequence of switches, which translates to sum of queue sizes (q1.qsize + ... + qn.qsize) along the path. Appendix D.2 includes more details on the translation between this interface and the syntax in Figure 4.

The throughput query (LS-T). Since there is no oversubscription in our leaf-spine network, we first ask whether the throughput between hosts 1 and 6 can drop below line-rate:

$$\begin{aligned}&\text{base\_wl} \rightarrow \forall t \in [10,10]: \ cenq(Out_6, t) < 5\\&\text{base\_wl} = (\forall t \in [1,10]: \ dst(In_1, t) = 6) \land\\&\qquad (\land_{i,j \in [1,6], i \neq j} \forall t \in [1,10]: dst(In_i, t) \neq dst(In_j, t))\end{aligned}$$

In_i and Out_i are the queues host i uses to send traffic into and receive traffic from the network, respectively. The query asks whether the total number of packets received by host i at time 10 is less than half of what it should have received at line rate. The base workload ensures that host 1 sends a steady stream

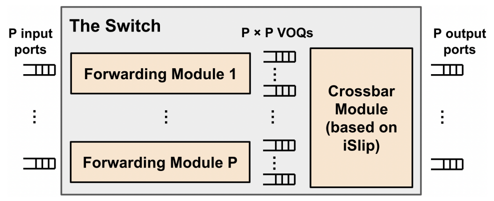


Figure 9: Modeling an input-queued switch with VOQs in FPerf.

of traffic to host 6 (at least 1 packet per time step), and no two hosts send traffic to the same destination so there is no traffic concentration at the hosts. For this query, FPerf finds the following workload:

$$\forall t \in [1,10]: \ dst(In_3,t) \geq 5 \wedge \forall t \in [1,7]: \ ecmp(In_1,t) = 1$$
$$\wedge \forall t \in [1,7]: \ ecmp(In_3,t) = 1 \wedge \forall t \in [1,10]: \ cenq(In_4,t) \leq 0.$$

That is, if there is another flow from host 3 to the third pod (constraint 1) with the same *ecmp* as the flow from host 1 to 6 (constraints 2 and 3), the two flows have to compete for bandwidth on the link from S_1 to L_3 (Figure 8) and host 6 will not receive traffic from host 1 at line rate. The last constraint ensures that the flow from host 3 has all the bandwidth from the second pod to itself to compete with the flow from host 1.

The latency query (LS-L). In the absence of queuing delay, it takes packets three time steps to go from a host in one pod to a host in a different pod. We want to know if it can take much longer, say, 13 time steps, for packets to go from host 1 to 6. To do so, we ask if it is possible to have a queue build up of at least 10 packets along the path of the flow:

$$\text{base\_wl} \rightarrow \forall t \in [10,10]: \ \Sigma_{q \in path}\, qsize(q,t) \geq 10$$
$$\text{base\_wl} = (\forall t \in [1,10]: \ dst(In_1,t) = 6) \wedge$$
$$(\forall t \in [1,10]: \ ecmp(In_1,t) = 1) \wedge (\forall t \in [1,10]: \ cenq(In_1,t) \leq t)$$

Here, the base workload ensures there is a flow from host 1 to 6 (constraints 1 and 2), and that the flow sends at most at line rate (constraint 3), so that there is no artificial queue build up from the flow's own packets. *path* is the set of queues the flow visits as it traverses L_1, S_1 (since in base_wl, $ecmp(In_t,t) = 1$), and L_3 (see Figure 8).

For this query, FPerf finds the following workload:

$$\forall t \in [1,8]: \ dst(In_3,t) = 6 \wedge \forall t \in [1,8]: \ ecmp(In_3,t) = 1$$
$$\wedge \forall t \in [1,10]: \ ecmp(In_5,t) = 6 \wedge \forall t \in [1,10]: \ cenq(In_4,t) \leq 0.$$

That is, if hosts 3 and 5 (from pods 2 and 3) send traffic to host 6 at the same time (constraints 1 to 3), there will be a queue build up of at least 10 packets along the path of the flow and its packets will experience high latency. Similar to the throughput query, the last constraint ensures that the flow from host 3 has all the bandwidth from the second pod to itself to contribute to quickly building up the queues.

Tractability. Tables 7b and 7a summarize statistics about different phases of workload synthesis for the queries about the leaf-spine network. The results are consistent with our observations in §7.3. Generating the good example set is expensive but there are opportunities for parallelization and further optimizations. The verification engine is efficient, verifying workloads in $< 1.7 sec.$ on average and $\sim 4 sec.$ in the worst case. Beyond what is described in §7.3, we employ other optimizations that contribute to this efficiency. Specifically, we use the forwarding rules in the leaf-spine topology to (1) remove certain VOQs from the switch crossbar if their corresponding input and output ports are not expected to communicate (e.g., a packet entering a leaf from a spine is expected to go to one of the output ports connected to the hosts and not other spines), and (2) constrain the values of the per-packet metadata to reduce the search space (e.g., all packets going into S_1 have *ecmp* set to 1). Finally, our search optimizations remain effective, avoiding calls to the verification engine in ~ 55 to 89% of the rounds, and the example traces effectively guide the search towards workloads that satisfy the query.

Takeaways. Queuing modules and their composition are expressive enough to model a variety of network components, from packet schedulers and classifiers to a network of switches. We did not need to make any changes to how we model contention points (§3) to model the leaf-spine network. Similarly, our workload synthesis techniques generalize beyond packet scheduling. Our example generation strategy and workload metrics need only minor changes to include the packet metadata needed for forwarding over the network, and our search algorithm can find workloads for the queries as effectively without any modifications.

9 Discussion and Future Directions

Scaling to large networks. Figure 7c shows how example generation, search, and workload verification times increase for the latency query on leaf- spine networks of increasing size. As the network size increases, the number of variables and constraints go from $45k$ and $46k$ to $182k$ and $181k$, and automated theorem provers (e.g., Max-SMT and SMT solvers) which we use in example generation and workload verification, can take exponentially longer as the problem size increases. For our largest evaluated network (not shown in the figure) with 4 spines, 4 leaves, and 16 servers (56 modules and 288 queues), it takes 224 minutes to find an answer.

There is room for more optimizations, some of which we have implemented for this experiment and discuss in Appendix D.3. But, as with any other approach that relies on similar formal methods tools, there is a limit to how many variables and constraints we can jointly reason about within a reasonable amount of time. As such, similar to data and control-plane verification tools that have matured over a decade to scale to large-scale networks, much work needs to be done to improve the scalability of formal methods tools for network performance analysis.

For example, we may need to develop new techniques, e.g., domain-specific algorithms for reasoning about network properties to replace SAT/SMT solvers [12, 48] or decompose global properties into local properties for modular analysis [15, 31]. Specifically, given that performance queries and properties such as latency lend themselves well to decomposition over regions and paths, we believe modular analysis to be key in scaling automated formal reasoning about performance and an important direction for future work.

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 [49], 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 opposed to the initial state. Finally, exploring verification techniques for reasoning about unbounded time [50] 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 FPerf to include packet sizes as extra variables, so they can be used in defining metrics and potentially reveal even more interesting workloads.

Generating traces from workloads. We can use the verification engine to generate example traces from the final workloads. Transforming these traces or the traces in G to concrete packets that can be injected into real-world networks (e.g., see Adapters in [37]) is an interesting future direction.

10 Related Work

Network Calculus. Network Calculus [6, 51] offers a uniform mathematical framework for analyzing performance guarantees. To use network calculus, one needs to model 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 then derive bounds on performance metrics such as throughput, latency, jitter, and loss [4, 52–54]. However, these curves need to be reasonably concise and provide tight bounds on the behavior of the network component and the input traffic pattern for the final bounds to be tight and useful. Deriving arrival and service curves is challenging, particularly for today's complex network functionality and traffic patterns [10].

Quantitative Reasoning. There is a line of work for reasoning about quantitative network properties: Some extend dataplane verifiers to reason about quantities such as link loads and hop counts [55–57]. Others reason about probabilistic aspects of networks (e.g., weighted ECMP) and answer probabilistic questions (e.g., the probability that packets reach a destination) [58,59]. Our work is similar in that performance properties are quantitative. However, these tools focus on *verification*, whereas, we propose to use *synthesis* to automatically generate not just one counter-example, but a workload that violates user-defined performance-related properties. Moreover, these tools abstract away many low level network details as nondeterministic, which we are able to model more precisely in SMT in our queueing modules.

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 [60–62]. Khan et al. propose to train Markov models that capture the temporal behavior and throughput and delay distributions of delay-based congestion control protocols [63]. 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 [64]. We propose generating workloads describing sets of traces in response to user-defined queries about performance problems.

Synthesis. Syntax-Guided Synthesis (SyGuS) is a general approach to program synthesis that uses a verifier together with a candidate program grammar. There are various ways to do SyGuS; there are enumerative [65, 66], stochastic [20, 30, 35], and logical approaches [67]. Our work is based on stochastic search. Moreover, recent work explores using synthesis in networking to generate packet processing code [19, 20], network configuration [21–23], configuration updates [24], or control-plane repairs [25, 26]. We use synthesis to generate workloads to reason about network performance.

11 Conclusion

Over the past decade, a large body of academic and industry work has demonstrated the feasibility and benefits of using formal methods to reason about the functional correctness of networks. Inspired by their success, we set out to bring the same benefits to analyzing network performance.

Along the way, we have developed efficient encodings of packet-level interactions that affect network performance. We have also found that when it comes to performance analysis, returning isolated packet traces that violate performance properties is not always useful. Instead, we argue that a more useful output is a workload that can concisely describe the commonality of a set of traces that can experience performance problems. We have shown how to apply existing synthesis techniques to generating such workloads and demonstrated the tractability of our approach using case studies.

This is only the start; as with other applications of formal methods to systems and networking, much work needs to be done to make such formal performance analysis approaches suitable for analyzing real-world networks, some of which we have outlined in this paper as future research directions.

Acknowledgments

We thank Laurent Vanbever, our shepherd, the anonymous reviewers, Jennifer Rexford, Shir Landau-Feibish, and Nate Foster for their helpful feedback. This work was supported in part by NSF grants CNS-2047283 and CNS-1704742, a Google faculty research award, and a Sloan fellowship.

References

[1] NS3 Network Simulator. https://www.nsnam.org/. Accessed: 09-2022.

[2] Mininet. http://mininet.org/. Accessed: 09-2022.

[3] 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 *ACM SOSP*, 2017.

[4] 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.

[5] Rayadurgam Srikant. *The mathematics of Internet congestion control*. Springer, 2004.

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

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

[8] 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.

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

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

[11] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *ACM SIGCOMM*, 2011.

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

[13] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

[14] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-Net: Real-time network verification using atoms. In *USENIX NSDI*, 2017.

[15] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, et al. Validating datacenters at scale. In *ACM SIGCOMM*. 2019.

[16] Announcing Network Intelligence Center – towards proactive network operations. https://cloud.google.com/blog/products/networking/announcing-network-intelligence-center. Accessed: 09-2022.

[17] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global WAN. In *ACM SIGCOMM*, 2020.

[18] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. GRoot: Proactive verification of DNS configurations. In *ACM SIGCOMM*, 2020.

[19] 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 *ACM SIGCOMM*, 2020.

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

[21] 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 *ACM SIGCOMM*, 2016.

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

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

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

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

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

[27] VMware to Advance Network Monitoring with Acquisition of Veriflow. https://blogs.vmware.com/management/2019/08/vmware-to-advance-network-monitoring-with-acquisition-of-veriflow.html. Accessed: 09-2022.

[28] Intentionet. https://www.intentionet.com/. Accessed: 09-2022.

[29] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. In *FMCAD*, 2013.

[30] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design*, 2016.

[31] Ryan Beckett and Ratul Mahajan. Capturing the state of research on network verification. https://netverify.fun/2-current-state-of-research/. Accessed: 09-2022.

[32] 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.

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

[34] Z3. https://github.com/Z3Prover/z3. Accessed: 09-2022.

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

[36] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Net2Text: Query-Guided summarization of network forwarding behaviors. In *USENIX NSDI*, 2018.

[37] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: closed-box analysis of network protocol implementations. In *ACM SIGCOMM*, 2021.

[38] Richard Uhler and Nirav Dave. Smten with satisfiability-based search. In *ACM OOPSLA*, 2014.

[39] Emina Torlak and Rastislav Bodik. A Lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN PLDI*, 2014.

[40] CBMC. https://www.cprover.org/cbmc/. Accessed: 09-2022.

[41] FPerf Github Repository. https://github.com/minmit/fperf. Accessed: 09-2022.

[42] Nvidia ConnectX SmartNICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/. Accessed: 09-2022.

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

[44] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *USENIX NSDI*, 2019.

[45] Apache Spark. https://spark.apache.org/. Accessed: 09-2022.

[46] Memcached: a Distributed Memory Object Caching System. http://www.memcached.org/. Accessed: 09-2022.

[47] Nick McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 1999.

[48] Todd Millstein. Toward modular network verification. https://netverify.fun/toward-modular-network-verification/. Accessed: 09-2022.

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

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

[51] Rene L Cruz. A calculus for network delay. parts I and II. *IEEE Transactions on Information Theory*, 1991.

[52] 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.

[53] 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.

[54] C-S Chang. Stability, queue length and delay. II. Stochastic queueing networks. In *IEEE Conference on Decision and Control*, 1992.

[55] Garvit Juniwal, Nikolaj Bjorner, Ratul Mahajan, Sanjit Seshia, and George Varghese. Quantitative network analysis. *Technical report*, 2016.

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

[57] 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.

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

[59] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. Bayonet: Probabilistic inference for networks. In *ACM SIGPLAN PLDI*, 2018.

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

[61] 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 *USENIX NSDI*, 2019.

[62] 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.

[63] 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. *ACM SIGCOMM Computer Communication Review*, 2021.

[64] Tomer Gilad, Nathan H Jay, Michael Shnaiderman, Brighten Godfrey, and Michael Schapira. Robustifying network protocols with adversarial examples. In *ACM HotNets*, 2019.

[65] Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed program synthesis. In *ACM SIGPLAN PLDI*, 2015.

[66] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In *ACM SIGPLAN-SIGACT POPL*, 2016.

[67] Sumit Gulwani and Ramarathnam Venkatesan. Component-Based synthesis applied to bitvector circuits. *Technical report*, 2009.

A 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, where each tuple represents a packet. This captures 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, where each tuple represents a packet. This 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. To model that, we take care of the dequeues first. We 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 and not any enqueues. Specifically, $tmp\_val[t][i]$ is true if the ith element of the queue will still contain a packet after the dequeues in that time step assuming no enqueues happen.

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 also 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.

Specifically, 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, suppose the queue has a packet at index i and no packets at index $i+1$. Then, there is a packet at any index $j \le i$ queue. Moreover, at any index $j > i$, the queue is empty. Specifically, for every $1 \le i < S$, we add:

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

B Defintion 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:

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

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, $\forall t \in [1,15] : cenq(Q1,t) \geq 2$, $\forall t \in [3,7] : cenq(Q2,t) \leq 5$, 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. So, $\mathsf{interval\_cnt}$ 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, more concise, and more interpretable.

C Details on Search Engine Optimizations

Reducing the search space. We have described one of our optimizations to reduce the search space in §6.5. Another optimization is detecting and ignoring "duplicate" workloads. 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 workloads and avoid generating them as candidates, reducing the space of workloads the search algorithm needs to explore.

Reducing calls to the verification engine. As we describe in §6.5, 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 a potentially expensive call to the verification engine if that workload comes up in the future.

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.

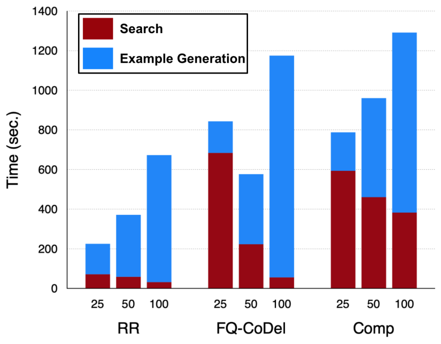


Figure 10: Search and example generations times for $|\mathsf{G}| = |\mathsf{B}| = 25$, 50, and 100. Prio results are not shown as the total time was less then $20s$ for Prio and would not be visible in the plot.

D More Details on the Leaf-Spine Case Study

D.1 Introducing New Packet Metadata

In the leaf-spine case study §8, packets have two metadata variables: dst, representing the final destination of the packet, and $ecmp$, with values in $[0,S]$ where S is the number of spines switches, representing the result of the ECMP hash of the packet's flow id modulo the number of spines.

We define the metrics $dst(q,t)$ as the destination of packets that enter q at time t and $ecmp(q,t)$ as the ECMP hash modulo number of spines for those packets. So, our workloads describe traffic patterns in which packets entering a queue at the same time have the same dst and $ecmp$.

For generating the base example, we add another optimization criteria, to maximize the "smoothness" of flows. That is, for the traffic entering from the hosts, the trace should not introduce new flows or go back and forth between flows with different dst and $ecmp$ if not needed for satisfying the query. When generating the rest of the good examples, we maintain the same "smoothness" criteria, and when minimizing the distance between eg_0 and eg_i, we include the difference in per-packet metadata in the computing the distance.

D.2 From User Interface to Logical Formulas

In the user interface described in §8, for the base workload, the users can provide a list of constraints using an interface that abstracts away the logical operators and expressions in Figure 4. Here, we describe how these constraints are translated to the logical formulas in Figure 4.

- [t1, t2] q.m $\oplus$ c becomes $\forall t \in [t_1,t_2] : m(q,t) \oplus c$.
- [t1, t2] (q1.m + ... + qn.m) $\oplus$ c becomes $\forall t \in [t_1,t_2] : \Sigma_{q\in\{q_1,\cdots,q_m\}} m(q,t) \oplus c$.
- [t1, t2] q1.m $\oplus$ q2.m becomes $\forall t \in [t_1,t_2] : m(q_1,t) \oplus m(q_2,t)$

- [t1, t2] (q1.m + ... + qn.m)/t $\oplus$ c becomes $\forall t \in [t_1, t_2]$: $\Sigma_{q \in \{q_1, \cdots, q_m\}} m(q,t) \oplus c \cdot t$

For queries, the users provide a list of questions with a similar interface, asking if the value of a metric over a time period for one or more queues can go above or below a threshold. A single question [t1, t2] lhs $\oplus$ rhs becomes $\exists t \in [t_1, t_2]$: trans(qlhs $\oplus$ rhs), where trans(qlhs $\oplus$ rhs) is the translation of the left hand side similar to what is described above. A list of questions will translate to the conjunction of their equivalent logical formulas. Note that queries in the form of $\forall t \in [t_1, t_2] lhs \oplus rhs$ are still possible in the user interface by creating a separate question for each time step between t_1 and t_2. It is also possible to extend the user interface to directly specify $\forall$ queries.

D.3 Example Generation Optimizations

For our scalability experiments in Figure 7c, we started with the default $|\mathsf{G}| = |\mathsf{B}| = 50$. However, example generation is expensive, and automated theorem provers (e.g., Max-SMT and SMT solvers) which we use in example generation and workload verification, can take exponentially longer as the problem size increases. So, to be able to observe the trends for larger networks, we used $|\mathsf{G}| = |\mathsf{B}| = 25$

We also employed extra optimizations when generating G. Recall that when generating the base example, we minimize the number of queues that have traffic in them. If a queue does not receive any traffic in the base example trace eg_0, it will stay empty in the rest of the traces in G. So, once eg_0 is generated, we create a "reduced" model in which remove the input queues that are marked as empty in eg_0 as they would be empty in the rest of the examples anyway. This helps reduce the number of variables and constraints, specifically in the crossbar modules of the leaf switches.

Moreover, recall that when generating trace eg_i, we constrain it to have different randomly chosen values from eg_{i-1} in $p = P$ random places. If no trace is found in two tries, we decrement p and try again. Instead of fixing the starting point to $p = P$, we set it to the moving average of the ps the worked when generating the last K examples.