A Layered Formal Methods Approach to Answering Queue-related Queries

Divya Raghunathan, Maria Apostolaki, and Aarti Gupta, *Princeton University*

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

A Layered Formal Methods Approach to Answering Queue-related Queries

Divya Raghunathan
Princeton University

Maria Apostolaki
Princeton University

Aarti Gupta
Princeton University

Abstract

Queue dynamics introduce significant uncertainty in network management tasks such as debugging, performance monitoring, and analysis. Despite numerous queue-monitoring techniques, many networks today continue to collect only per-port packet counts (*e.g.,* using SNMP). Although queue lengths are correlated with packet counts, deriving the precise correlation between them is very challenging since packet counts do not specify many quantities (*e.g.,* packet arrival order) which affect queue lengths.

This paper presents QUASI, a system that can answer many queue-related queries using only coarse-grained per-port packet counts. QUASI checks whether there exists a packet trace that is consistent with the packet counts and satisfies a query. To scale on large problem instances, QUASI relies on a layered approach and on a novel *enqueue-rate* abstraction, which is lossless for the class of queries that QUASI answers. The first layer employs a novel and efficient algorithm that generates a *cover-set* of abstract traces, constructs *representative* abstract traces from the cover-set, and efficiently checks each representative abstract trace by leveraging a known result on $(0,1)$-matrix existence. The first layer guarantees no false negatives: if the first layer says "No", there is no packet trace consistent with the observed packet counts that makes the query true. If it says "Yes", further verification is needed, which the second layer resolves using an SMT solver. As a result, QUASI has no false positives and no false negatives.

Our evaluations show that QUASI is up to 10^6X faster than state-of-the-art, and can answer non-trivial queries about queue metrics (*e.g.,* queue length) using minute-granularity packet counts. Our work is the first step toward more practical formal performance analysis under given measurements.

1 Introduction

Many challenging network management tasks can be reduced to queue-related queries, as queuing delays often represent the most significant source of uncertainty for a packet on its journey [21]. For instance, a network operator can verify a latency Service Level Objective (SLO) for a path by querying the bounds for each queue's length on that path. Similarly, a decision about whether to increase the size of on-chip buffers or whether to allow certain applications to share a switch could be informed by querying how often multiple queues of the same device experience bursts concurrently [1, 2].

While there are various techniques for measuring queue behavior, fine-grained queue monitoring is often costly to collect, store, and analyze, and it may even require specialized hardware [6, 9, 16, 18, 20]. As a result, commodity data centers, ISPs, and enterprise networks often rely on simpler, always-on packet count monitors that track the number of incoming and outgoing packets at each port (*e.g.,* SNMP [7, 12, 19, 23, 25, 26]). These per-port packet counts are highly scalable and are typically collected at coarse-grained intervals, such as every few minutes. They are useful for estimating average utilization and identifying traffic patterns, but the widely held assumption is that they cannot be used for answering queue-related queries.

Upon closer examination, this assumption appears flawed. Per-port packet counts are actually closely correlated with queue lengths and can impose significant constraints on queue dynamics, hence helping answer queue-related queries. For instance, multiple packets are required for a queue to form, and bursts only occur when multiple ports simultaneously send to the same queue (fan-in). Additionally, there should be a limit to the number of concurrent bursts a symmetric device can handle. These observations motivate the question this paper seeks to answer: *Can we answer queue-related queries using only coarse-grained packet counts?*

At first glance, this task appears impossible. The first hurdle lies in the discrepancy between the query metric (queue length) and the available data (per-port packet counts). Although these two are closely related, establishing a closed-form relationship between them is not straightforward. The counts only provide the total number of packets received and sent at each port, without information on their timing, sequences, or routing (i.e., which input port's packet was forwarded to which output port and when). This information defines a *packet trace* and are crucial for deducing queue lengths. Hence, to answer queries about queue lengths, one would need to consider a vast number of packet traces, identify those consistent with the given per-port counts, and then determine the answer based on these traces. The search space expands significantly as the monitoring interval duration increases. An alternative approach would be to leverage FPerf [4], a system that models network components and their queues in logic, and synthesizes workloads that satisfy a given query. FPerf solves a different performance analysis problem, so we modified FPerf to check if there exists a packet trace consistent with given measurements (specified as constraints over a workload) that satisfies a given query. However, FPerf does not support output and dropped packet count measurements, and quickly becomes unscalable (§7.4).

Layered approach. To navigate the search space of packet traces more efficiently, we have developed a layered approach called QUASI (**Qu**eue **A**nalysis from **S**NMP **I**nformation), drawing inspiration from the abstraction-refinement paradigm [10], which has seen tremendous success in the formal methods community [8, 17]. At its core, QUASI provides Boolean answers to queries (*e.g.,* answer yes or no: could this queue have n packets?). QUASI can also handle quantitative queries by running multiple Boolean queries (*e.g.,* to answer: what is the maximum length this queue could have reached?). The first layer of QUASI performs an efficient over-approximate analysis using representations of packet traces as $(0,1)$-matrices, such that a negative result is conclusive and does not require further analysis – we prove that QUASI's first layer will not incorrectly answer a query negatively. If this first layer is inconclusive, QUASI progresses to a second layer that utilizes an exact but less scalable analysis based on use of a Satisfiability Modulo Theories (SMT) solver [5, 11]. This layered strategy allows QUASI to balance efficiency and accuracy, reserving the use of more computationally intensive methods for situations where they are genuinely necessary.

Novel abstraction. QUASI is centered around a novel, powerful, and lossless abstraction that we call *enqueue-rate*. The enqueue-rate abstraction enables QUASI to reason about multiple packet traces simultaneously, thereby improving efficiency without any loss of accuracy. This abstraction is based on the insight that the exact routing of packets between input and output ports is not relevant for answering the queue-related queries supported by QUASI.

Efficient over-approximate analysis. QUASI's first layer leverages the enqueue-rate abstraction and incorporates three key innovations that result in high efficiency. First, QUASI derives necessary conditions from the query, represented as a disjunction of concise representations of sets of abstract traces; each set can be investigated independently or in parallel with others. Second, we show that for each such set of abstract traces, we can construct a special abstract trace — termed the *most-uniform* abstract trace — that serves as a representative of all abstract traces in the set. In particular, we prove that it is enough to check just the representative abstract trace for each set. Finally, we show that we can efficiently check each representative abstract trace using a known result from combinatorics, specifically the Gale-Ryser theorem [13, 24].

Exact Analysis. QUASI's second layer also uses the enqueue-rate abstraction, using an SMT solver to precisely model the *exact* (rather than necessary) conditions for an abstract trace to be consistent with the given measurements and the query.

Summary of results. We evaluate QUASI on simulated data and compare it against a heuristic baseline and FPerf [4] on a variety of queries (both Boolean and quantitative). Our results show that QUASI is useful: QUASI's first layer finds upper bounds on maximum queue length and buffer occupancy up to 58% tighter than bounds found by a heuristic analysis within one second; QUASI's second layer finds the exact value in 25 minutes. QUASI is 10^6x faster than FPerf and supports an important class of burst-related queries that FPerf does not support. QUASI scales to realistic monitoring intervals (5 mins/300 million time steps) for queue length queries.

2 Motivation

In this section, we start by describing use cases that showcase the usefulness of provably answering queue-related queries, especially when done with coarse-grained per-port packet count measurements. Then, we explain the shortcomings of existing approaches to measuring and inferring queue dynamics.

2.1 Why are Queue-related Queries Useful?

Having visibility on queue lengths in a network (through direct or indirect measurements) is critical for multiple network management tasks. We discuss use cases related to root-cause analysis, provisioning, and performance guarantees.

First, visibility over queue lengths makes debugging network incidents easier. Consider, for example, an operator debugging an incident of packet drops on a particular network device. The operator would benefit from learning whether the drops could have been caused by buffer pressure (*i.e.,* excessive buffering on ports distinct from the one dropping), or whether they could have been from a bursty application. The operator could learn this by looking at the queue-length time series of each queue on the device. Yet access to such data via monitoring alone would require an always-on tool that monitors all queues of all devices at a millisecond (at least) granularity. Considering the cost, a tool that would *provably* answer at least some such queries using cheap coarse-grained packet counts would be useful.

Second, visibility over queue lengths simplifies network planning. Consider an operator that aims to place applications in their network avoiding bursty applications sharing the same device, or an operator investigating whether they need to upgrade on-device buffers. Observe that such a query needs to run over historical queue length data spanning a good amount of time to contain various applications. Even if these data were collected in the past at ms granularity, it is unlikely that they would be currently available. Hence, a tool that would "piggyback" queries about queues on coarse-grained packet counts that are used for other tasks (*e.g.,* predicting utilization, traffic matrices) and hence are more likely to be available, would be very useful.

Finally, answering queue-related queries could be useful for verification. Queue management in high-speed switches involves intricate mechanisms for flow control across multiple queues on different ports. Implementation errors by equipment manufacturers can result in counterintuitive outcomes, such as high packet loss and delays even during periods of low link utilization. Investigating whether the device's working is consistent with the specification is very challenging. Accurate and fine-grained queue length monitoring would be useful but might not be reliable. Hence, it would help to cross-check those measurements with coarse-grained packet counts. In this case, a tool that can provably answer queries about what queue length values are possible given the packet counts would be useful.

2.2 Limitations of Inferring Queue Lengths

Directly measuring and storing queue lengths at a fine-grained granularity (e.g., nanosecond level) in an always-on manner is impractical [15], leading to the development of various inference methods such as Zoom2Net [15] and SIMON [14]. While these methods are innovative in that they do not require additional resources or hardware, they lack correctness guarantees. For example, Zoom2Net generates fine-grained telemetry guaranteed to satisfy selected constraints enforcing consistency with coarse-grained measurements and switch operation. However, these constraints are not sufficient, so the generated fine-grained telemetry might be impossible in practice. Also, it cannot prove, *e.g.,* that a certain queue length could not have occurred. This limitation becomes particularly problematic when an operator needs to understand the worst-case scenario or is concerned about a specific set of scenarios with some property. Consider an operator troubleshooting an incident of increased end-to-end latency: knowing the worst-case queuing delay for each device on-path (or that the delay has not exceeded some threshold) would help her safely narrow down the search to a smaller set of devices. However, if the inferred queueing delay is an underestimate, it can mislead the operator, complicating the debugging process. Beyond these limitations, both works assume access to coarse-grained measurements of queue length/delay. Zoom2Net considers periodic and maximum queue lengths (along with packet counts), and SIMON assumes end-to-end queueing delay.

3 Overview of QUASI

After formally defining our problem (§3.1), we introduce a running toy example that shows how leveraging coarse-grained packet counts can be useful in deducing queue dynamics

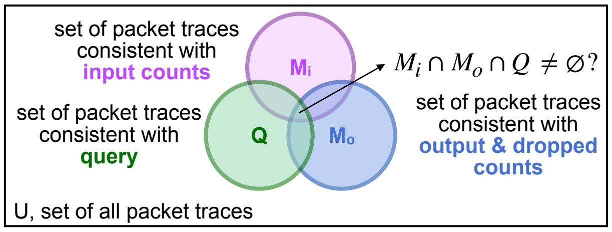

set of packet traces consistent with input counts
set of packet traces consistent with query
set of packet traces consistent with output & dropped counts
U, set of all packet traces
M_i
Q
M_o
$M_i \cap M_o \cap Q \neq \varnothing$?

Figure 1: We recast the problem of answering a queue-related query under given measurements into the problem of checking whether the intersection of M_i, M_o, and Q is non-empty.

(§3.2). More importantly, the example showcases that even in such a seemingly simple example, answering a queue-related query is nontrivial. We use this example to illustrate the key steps in how QUASI answers queue-related queries in §4.4, after we have introduced key concepts in §4.3.

3.1 Problem Definition

We are given packet counts of received (input), sent (output), and dropped packets for each port of a switch, aggregated over a monitoring interval. We are also given a query describing an incident/property or value depending on one or more queues during the interval which can be answered positively or negatively.

Goal. Our goal is to build a system that can answer positively if there exists some sequence of packets (called a *packet trace*) which, if arrived at the switch during the interval, would be consistent with input, output, and dropped counts, and would satisfy the query. The system should answer negatively if such a packet trace does not exist. The exact syntax of the query language is shown in Fig. 4 (§4.2).

More concretely, we formulate the problem of answering a queue-related query as determining whether the intersection of three sets is non-empty, as shown in Fig. 1:

M_i : set of all packet traces consistent with the per-port input counts of all ports of a switch.

M_o : set of all packet traces consistent with the per-port output and dropped counts of all ports of a switch.

Q : set of all packet traces that make the query true.

There exists a packet trace consistent with the measurements that makes the query true if and only if $M_i \cap M_o \cap Q \neq \emptyset$.

3.2 Motivating Example

Consider a cloud operator who receives a complaint from a customer claiming that the latency between her silicon servers violated the service level objective (SLO) at some point between 9 am and 10 am on a given day. Since the customer's servers are connected through a single switch (shown in Fig. 2), the operator deduces that the only way the cloud provider might be liable is if the queueing latency at the switch exceeded a threshold close to the SLO during that time. Consider that the operator lacks access to fine-grained queue-length measurements, either because the hardware

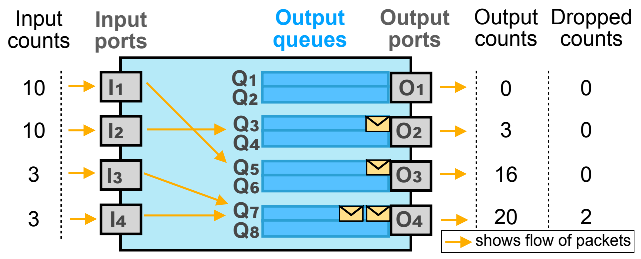


Figure 2: Output-queued switch with per-port packet counts.

does not support such fine-grained monitoring or because collecting and storing these measurements is only available on-demand due to its high cost. Instead, the operator has access to SNMP [12] packet counts *i.e.,* per-port packet counts aggregated over a monitoring interval of 20 time steps, where each time step represents either a packet received or sent.[1]

Fig. 2 shows an output-queued switch with 4 ports. Packets are received through input ports, forwarded to output queues, and sent through output ports. Each port receives/sends at most one packet in a time step, and multiple input ports can simultaneously forward packets to the same output queue. We assume packet forwarding takes negligible time. Packets are dropped if the destined queue is at its maximum size (10 packets in our example). Fig. 2 shows per-port input, output, and dropped counts, *i.e.,* the number of packets received, sent, and dropped respectively during a single interval of 20 time steps. Note that queues could store packets at the start of the monitoring interval, so the total count of sent and dropped packets can exceed the total count of received packets. The operator's query is: "Could port O_2's queues have a combined queue length of 7 packets at any time during this interval?" The operator would need to ask this query for every interval between 9 and 10 am. Next, we focus on the query for a single interval. The query is expressed in our query language as

$$\exists t \in [0,T].\ \sum_{q \in O_2} qlen(q,t) \geq 7 \tag{1}$$

A naive observer would answer the query positively. If one were to consider port O_2 in isolation from the rest of the ports, one might think that there is a packet trace which results in a queue length of 7 packets and is consistent with its packet counts during the interval. Fig. 5 shows such a packet trace P_1, represented as a 2-D matrix with $P_1[q][t]$ showing the packets enqueued at queue q at time t, with packet colors denoting the input port each packet came from; blank entries denote no packets enqueued.

In this packet trace, O_2 receives no packets until the last 3 time steps, when it receives a sequence of 4, 4, and 2 packets. This causes the queue length to increase by 3, 3 and 1 in each time step since the port dequeues one packet per time step and the remaining packets are queued. Assuming port O_2 is initially empty, at the end of the interval, it has $3+3+1=7$ packets queued, satisfying the query. In addition, since O_2 receives packets only in the last 3 time steps, it dequeues a total of 3 packets during the interval, matching its sent count. Since port O_2 never reaches the maximum queue size of 10, it does not drop any packets, thereby matching its dropped count. However, this naive reasoning is erroneous.

The correct answer to the query is negative. In fact, there is no packet trace consistent with the observed measurements that will make the query true, *i.e.,* O_2's queue length is always less than 7 during the monitoring interval. The reason that the correct answer is negative though it appears positive at first glance is that the packet counts of other ports constrain the number of packets that could have been enqueued at O_2.

At a high level, answering queue-related queries is challenging because the queries involve quantities that are not directly measured and cannot be inferred by a simple analysis of the measurements. While a query metric (*e.g.,* queue length) is correlated with measured signals (packet counts) it also depends on information absent in packet counts, namely on the arrival sequence of packets entering the switch, and to which output queue each packet is forwarded.[2] This information is present in a packet trace, which specifies which port's packet was forwarded to which port and when. However, a huge number of packet traces could be consistent with given packet counts, and reasoning about all these packet traces is computationally expensive.

[1] we have scaled down the interval to make the example easier to follow, but we have evaluated QUASI on an interval with 300 million time steps (§7.1).

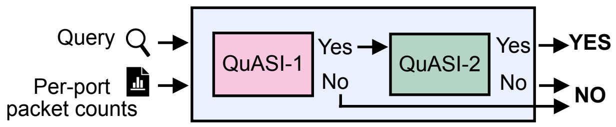


Figure 3: QUASI architecture.

4 QUASI Design

This section describes the key concepts QUASI relies on and demonstrates their usefulness on the running example (§3.2).

4.1 QUASI's Layered Architecture

Due to the scalability limitations of an exact analysis, we use a layered approach consisting of two layers, QUASI-1 and QUASI-2, as shown in Fig. 3. QUASI-1 is a fast, over-approximate analysis which is guaranteed to never report a false negative. Hence, if QUASI-1 answers "No", QUASI reports "No" and terminates. If QUASI-1 answers "Yes", it could be a false positive, so the analysis proceeds to QUASI-2 which is a more expensive, exact analysis that reports the correct answer. QUASI's first layer is accurate enough to conclusively answer non-trivial queries in a fraction of a second, as we demonstrate in the evaluations (§7). By running the more expensive

[2] For context, Zoom2Net [15] does not deal with this challenge – it assumes there is a coarse-grained version of the signal of interest.

$$
\begin{aligned}
query &::= \{\exists|\forall\}\ t \in [T_1, T_2].\ cmp \\
cmp &::= lhs\ ?\ rhs \mid \bigwedge lhs\ ?\ rhs \\
lhs &::= \{\exists|\forall\} q \in Qs.\ m(q,t) \mid \textstyle\sum_{q \in Qs} m(q,t) \\
&\quad \mid m(Q_1,t) - m(Q_2,t) \\
m(q,t) &::= enq(q,t) \mid cenq(q,t) \mid qlen(q,t) \\
rhs &::= K \mid K.t
\end{aligned}
$$

Figure 4: Query language. T_1, T_2, K are constants, ? denotes a comparison operator, and Q_1, Q_2 denote queues in a switch.

second layer only on queries that cannot be answered conclusively by the fast first layer, QUASI can scale to *tens of thousands* of time steps. This is well beyond the capabilities of existing tools [4], which take several hours even for 10 time steps.

4.2 Query Language

A query in QUASI is a comparison (or conjunction of comparisons) involving a metric over one or more queues in a switch; Fig. 4 shows the syntax. We consider metrics (1) $enq(q,t)$: number of packets received by queue q at time step t, (2) $cenq(q,t)$: total number of packets received by queue q until time step t, and (3) $qlen(q,t)$: queue length (in packets) at queue q at time step t. Queries allow quantification over (bounded) time steps as well as queues. Our query language is expressive enough to allow interesting questions related to bursts, queue lengths, and buffer occupancy, including some queries that FPerf [4] does not support, *e.g.,* the Burst query used in our evaluation (§7). Our query language can be extended with interfaces to languages like Python or SQL to make it easier to use for operators unfamiliar with logic.

QUASI currently supports single-switch queries, but can also be used to answer some queries over multiple switches. For example, we can find the maximum queueing latency along a given network path using QUASI to find the maximum queueing latency at individual switches on the path and adding up these values. Using only QUASI-1, the answer will be an upper bound on the maximum latency along the path while QUASI-2 will report the true maximum latency.

4.3 Key Concepts

We present key concepts required to understand our approach.

Packet trace. A packet trace P is a $N_q \times T \times N$ $(0,1)$-matrix with entry $P[q][t][i]$ equal to 1 if input port i forwards a packet to queue q at time step t, and 0 otherwise. N denotes the number of switch input ports, N_q denotes the total number of output queues, and T denotes the number of time steps in the monitoring interval. Fig. 5 shows packet trace P_1 as a 2-D matrix with packet color denoting the input port it came from; blank entries are 0.

Enqueue-rate abstraction. We target queries that do not distinguish between packets in a queue based on their origin input ports. Thus, it is safe to ignore input port information in a packet trace. Our enqueue-rate abstraction defines an *abstract* packet trace that represents the *enqueue-rate*, the number of packets enqueued to each queue at each time step, ignoring the input port each packet came from. An abstract trace represents a set of packet traces, one packet trace for each labelling of packets with input ports. QUASI-2 uses the enqueue-rate abstraction to reduce the number of variables in its SMT formulation without any loss of precision due to abstraction, since our query language (Fig. 4) treats all packets in a queue as identical.

Abstract trace. An abstract trace A is a $N_q \times T$ matrix with entry $A[q][t]$ denoting the number of packets enqueued to queue q at time step t. An abstract trace represents a set of packet traces. We say packet trace P is "contained" in A iff $\forall q, t.\ \sum_{1 \le i \le N} P[q][t][i] = A[q][t]$. Fig. 5 shows an abstract trace A_1 which contains both packet traces P_1 and P_2, which differ in the input ports of some packets but have the same number of packets enqueued at each queue at each time step.

Cover-set F_C. In QUASI-1, we use the enqueue-rate abstraction at a coarser granularity than in QUASI-2. The first module of QUASI-1 generates a cover-set $F_C = F_C^1 \vee \ldots \vee F_C^k$, a finite disjunction of *components* $F_C^1, \ldots, F_C^k$, formulas representing *sets* of abstract traces. Each component F_C^j is a conjunction of constraints involving enqueue-rate (*enq*); we do not enumerate the abstract traces satisfying F_C^j. A cover-set represents all packet traces consistent with the query and output counts and dropped counts, *i.e.,* $M_o \cap Q \subseteq C$, where C denotes the set of packet traces satisfying the cover-set formula F_C. Fig. 5 shows the cover-set that QUASI generates for our motivating example; it has only one component F_C^1 (*i.e.,* $k = 1$).

Representative abstract trace. Each (non-empty) component F_C^j in a cover-set F_C has a representative abstract trace R_j that is "most-uniform" in the set of abstract traces C_j satisfying F_C^j. We prove that if R_j does not contain a (concrete) packet trace consistent with input counts, then the entire C_j has no packet trace consistent with input counts (Theorem 5.3). Representative abstract traces provide a correctness-preserving *reduction* in the search space: it is enough to consider only the representative R_j in each component. The second module of QUASI-1, MUC *constructs* a representative abstract trace R_j for a given F_C^j. For our motivating example, Fig. 5 shows the representative abstract trace R_1 for F_C^1.

Abstract trace consistency. We say abstract trace A is consistent with input counts (or is *input-consistent*) if there exists a labeling function *Lbl* that maps each packet in A to an input port such that: (1) packets at the same time step are assigned different input ports (an input port can forward at most one packet in a time step), and (2) each input port label i is used exactly $rcv[i]$ times, where $rcv[i]$ is port i's input count. We efficiently check input-consistency of each representative abstract trace by reducing it to checking the existence of a $(0,1)$-matrix with given row sums and column sums (Theorem 5.4) and using a known combinatorics result (Gale-Ryser theorem [13, 24]).

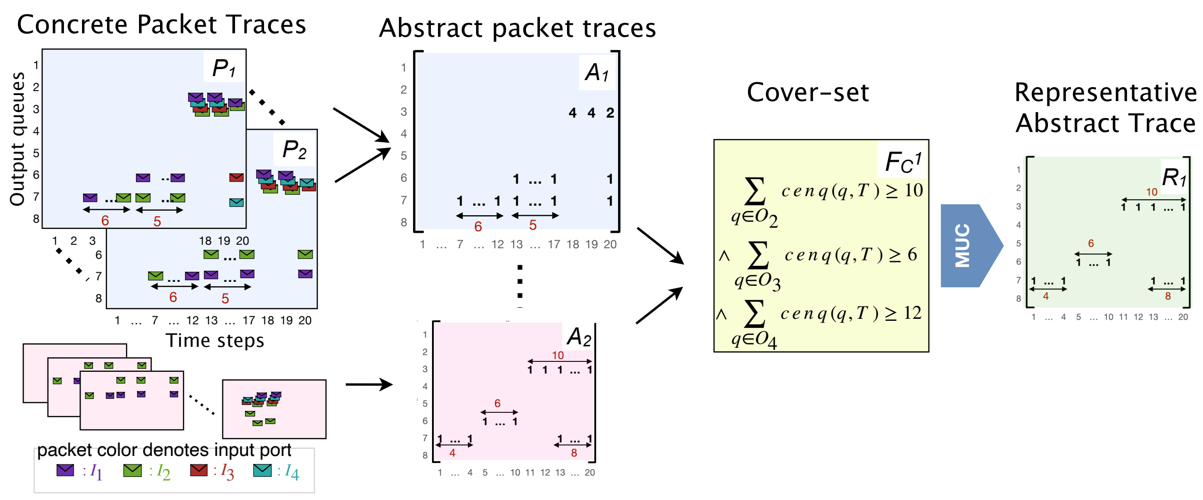

Concrete Packet Traces
Abstract packet traces
Cover-set
Representative Abstract Trace
Output queues
Time steps
MUC
packet color denotes input port
I_1 I_2 I_3 I_4
P_1 P_2 A_1 A_2 $F_C{}^1$ R_1
$\sum_{q\in O_2} cenq(q,T) \geq 10$
$\wedge \sum_{q\in O_3} cenq(q,T) \geq 6$
$\wedge \sum_{q\in O_4} cenq(q,T) \geq 12$

Figure 5: Multiple concrete packet traces (*e.g.,* P_1 and P_2) are mapped through the enqueue-rate abstraction to an abstract trace (*e.g.,* A_1). Multiple abstract traces (*e.g.,* A_1 and A_2) are represented by a cover-set (*e.g.,* F_C^1 with number of components $k = 1$). The MUC constructs a most-uniform abstract trace which can be used as a representative in F_C^1 for checking input-consistency. For their analysis, QUASI-1 uses cover-sets and representative abstract traces, while QUASI-2 uses abstract traces.

4.4 QUASI on Motivating Example

We illustrate the key steps of QUASI on our motivating example (§3.2). While QUASI is composed of two layers, it reaches a conclusive negative answer already at the first layer, which is guaranteed to be correct. QUASI's first layer generates a formula F_C (the cover-set) that represents a set C containing all packet traces consistent with output counts, dropped counts, and the query, *i.e.,* $M_o \cap Q \subseteq C$.

$$F_C = \sum_{q\in O_2} cenq(q,T) \geq 10 \wedge \sum_{q\in O_3} cenq(q,T) \geq 6$$
$$\wedge \sum_{q\in O_4} cenq(q,T) \geq 12 \qquad (2)$$

Eqn. (2) shows F_C in terms of the cumulative enqueue-rate $cenq(q,t)$, the number of packets enqueued at a queue q until time step t. F_C states that during the monitoring interval, output port O_2 must receive at least 10 packets, output port O_3 must receive at least 6 packets and output port O_4 must receive at least 12 packets.

Next, QUASI needs to determine whether each component F_C^i in F_C is consistent with input counts (here, $F_C = F_C^1$ with $k = 1$), *i.e.,* whether $F_C^1 \cap M_i = \emptyset$. Note that F_C^1 itself represents a large set of packet traces, so checking each individual packet trace is impractical. QUASI constructs one representative abstract trace, denoted R_1, which satisfies F_C^1. R_1 specifies the number of packets enqueued over time at each queue (or each port, depending on the query). We prove that it suffices to check *only* R_1 for consistency with input counts: if R_1 does not contain a packet trace that is input-consistent, then no other abstract trace satisfying F_C^1 will contain a packet trace that is input-consistent (Theorem 5.3). Fig. 5 shows R_1 for our example.

To check R_1 for consistency with input counts, QUASI first checks if the total number of packets in R_1 exceeds the total number of packets received by the switch. If yes, R_1 is inconsistent with input counts; otherwise, QUASI proceeds to run an efficient matrix-based consistency check. For our example, the total number of packets in R_1 is 28, which is more than the total number of packets received (26). Hence, QUASI answers the query negatively, which is the correct answer for this example.

5 QUASI: Layer 1

QUASI-1 consists of three modules (Fig. 6):

1. **Cover-set generator (CSG)** generates a cover-set $F_C = F_C^1 \vee \ldots \vee F_C^k$ which represents all packet traces consistent with output and dropped counts that make the query true, *i.e.,* $M_o \cap Q \subseteq C$, where C denotes the set of packet traces satisfying F_C. Fig. 6 shows C with $k = 4$ as a yellow circle superimposed on a Venn diagram showing the sets M_i, M_o, and Q from our problem formulation (§3.1).
2. **Most-uniform Abstract Trace Constructor (MUC)** constructs a representative abstract trace R_j (shown as a small red square in Fig. 6) that is *most-uniform* for each component F_C^j. We prove that it is enough to check *only* each R_j for consistency with input counts (Theorem 5.3).
3. **Matrix-based Consistency Checker (MCC)** determines whether a given representative abstract trace contains a packet trace that is input-consistent, *i.e.,* whether R_j intersects with M_i. We reduce this problem to checking the existence of a $(0,1)$-matrix with given row sums and column sums (Theorem 5.4) and use the Gale-Ryser theorem [13, 24] to solve it efficiently.

If some representative abstract trace R_j is found to be consistent with input counts, QUASI-1 answers the query positively. QUASI-1 answers the query negatively if every R_j is inconsistent with input counts.

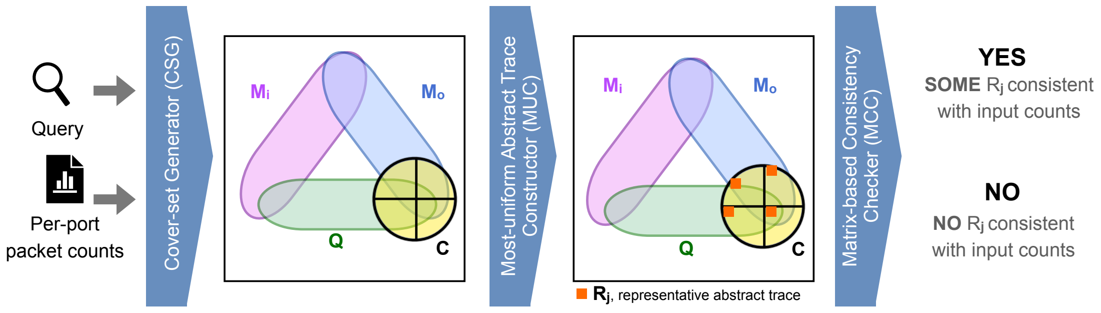

Query
Per-port packet counts
Cover-set Generator (CSG)
Mi
Mo
Q
C
Most-uniform Abstract Trace Constructor (MUC)
Rj, representative abstract trace
Matrix-based Consistency Checker (MCC)
YES
SOME Rj consistent with input counts
NO
NO Rj consistent with input counts

Figure 6: QUASI-1 design: (1) CSG generates a cover-set formula F_C representing a set $C=C_1\cup\ldots\cup C_4$, with $C\supseteq M_o\cap Q$, (2) MUC constructs a representative R_j for each C_j, (3) MCC checks if $R_j\cap M_i=\emptyset$ for each R_j; if true, QUASI-1 reports "No", otherwise "Yes".

Theorem 5.1 (QUASI-1 correctness). *If* QUASI-*1 answers a query negatively, there is no packet trace consistent with the measurements that makes the query true,* i.e., QUASI-*1 will never report a false negative.*

Proof. Assume QUASI-1 answers a query Q negatively. This means the MCC determined that each representative abstract trace R_j (for C_j) does not contain a packet trace that is consistent with input counts, *i.e.,* $R_j\cap M_i=\emptyset$. For each C_j, by Theorem 5.3, no other abstract trace in C_j is consistent with input counts, *i.e.,* $C_j\cap M_i=\emptyset$. Hence, there is no packet trace in $C=C_1\cup\ldots\cup C_k$ which is consistent with input counts, *i.e.,* $C\cap M_i=\emptyset$. By Lemma 5.2, C contains all packet traces consistent with output and dropped counts that make the query true, *i.e.,* $M_o\cap Q\subseteq C$. Since $C\cap M_i=\emptyset$, it follows that $M_o\cap Q\cap M_i=\emptyset$. Hence, there exists no packet trace consistent with the measurements that makes the query true. □

Our proof depends on Lemma 5.2 and Theorem 5.3 – these are discussed in the related subsections.

False positives. If QUASI-1 answers a query positively, it is possible that no trace exists which is consistent with the measurements and query. This is because the cover-set generated by the CSG is an *over-approximation*, *i.e.,* it could include additional packet traces not consistent with output and dropped counts and/or the query. If one of these extra packet traces is consistent with input counts then QUASI-1 could incorrectly answer the query positively.

5.1 Generating a Cover-set

The CSG constructs symbolic representations (as formulas) of two sets: (1) Q', which contains all packet traces satisfying the query ($Q\subseteq Q'$) and (2) M'_o, which contains all packet traces consistent with output and dropped counts (*i.e.,* $M_o\subseteq M'_o$). It returns $F_C=F_C^1\vee\ldots\vee F_C^k$, obtained by performing standard syntactical transformations on the formula for $M'_o\cap Q'$.

M'_o, necessary conditions for consistency with output and dropped counts. Each output port O_i must receive a minimum number of packets (across all its output queues) during the monitoring interval to be consistent with its output count (deq_i) and dropped count (drp_i).

$$\forall 1\le i\le N.\ \sum_{q\in O_i} cenq(q,T)\ge deq_i+drp_i-ilen\_ub_i \quad (3)$$

$ilen\_ub_i$ is an upper bound on $ilen_i$, the number of packets queued at port O_i at the start of the interval. Note that $ilen\_ub_i\le deq_i$ otherwise port O_i would dequeue more than its output count. Also, $ilen\_ub_i\le L$, the maximum number of packets that can be queued at a port. Hence, we set $ilen\_ub_i=min(L,deq_i)$. We can improve this bound further depending on the query, *e.g.,* if the query asks if the queue length can be always below some K.

Equation (3) is obtained from conservation of packets at an output port, assuming that output and dropped counts hold. By conservation of packets, the number of packets queued in the port at the end of the monitoring interval ($flen_i$) is equal to the difference between the number of packets received (or were queued initially) at the port during the interval and the number of packets that left the port (or were dropped) during the interval. Formally,

$$flen_i=ilen_i+cenq(O_i,T)-cdeq(O_i,T)-cdrp(O_i,T) \quad (4)$$

where $cenq(O_i,\ T)$, $cdeq(O_i,\ T)$ and $cdrp(O_i,\ T)$ denote the total number of packets enqueued, dequeued, and dropped respectively at port O_i from time step 0 until T. By consistency with output counts, $cdeq(O_i,T)=deq_i$; by consistency with dropped counts, $cdrp(O_i,T)=drop_i$. Using $flen_i\ge 0$ and rearranging, we get $cenq(O_i,T)\ge deq_i+drp_i-ilen_i\ge deq_i+drp_i-ilen\_ub_i$. Observing that $cenq(O_i,T)=\sum_{q\in O_i}cenq(q,T)$, we get eqn. (3).

Q', necessary conditions to satisfy query.

Case 1: Query metric is *enq* or *cenq*. For queries involving the metrics enqueue-rate (*enq*) or cumulative enqueue-rate (*cenq*) the query Q is translated into an equivalent quantifier-free formula by eliminating quantifiers over queues and time steps to get Q', which is semantically equivalent to Q. Existential quantifiers are eliminated using disjunctions and universal quantifiers are eliminated using conjunctions.

Case 2: Query metric is *qlen*. Our necessary conditions need to use a different metric (*enq* or *cenq*) from the metric in the query, so we cannot obtain Q' just by eliminating quantifiers as in the previous case. To relate queue length to enqueue-rate, we use conservation of packets (5) at the queue(s) in the query, setting $qlen(q,t)$ to an appropriate value depending on the query. We explain how we construct our necessary conditions for the query $QgeK : \exists t \in [st, end].qlen(q,t) \geq K$ which asks if queue q can have at least $K > 0$ packets at some time step $t \in [st, end]$; other queries are handled similarly.

By conservation of packets, the number of packets in the queue at time step t ($qlen(q,t)$) is equal to the difference between the number of packets received (or were queued initially) at the queue and the number of packets that left the queue (or were dropped) until t:

$$qlen(q,t) = il(q) + cenq(q,t) - cdeq(q,t) - cdrp(q,t) \quad (5)$$

where $il(q)$ denotes the number of packets in the queue at the start of the monitoring interval; and $cenq(q,t)$, $cdeq(q,t)$, $cdrp(q,t)$ denote the number of packets enqueued, dequeued, and dropped, respectively, at queue q from time step 0 until t. For the query to be true, $qlen(q,t) \geq K$ for some $t \in [st, end]$; substituting this in eqn. (5) we get

$$il(q) + cenq(q,t) - cdeq(q,t) - cdrp(q,t) \geq K$$
$$cenq(q,t) \geq K - il(q) + cdeq(q,t) + cdrp(q,t)$$

We replace each term on the right-hand-side by a lower/upper bound as appropriate: an upper bound for $il(q)$, and lower bounds for $cdeq(q,t)$ and $cdrp(q,t)$.

Upper bound on initial queue length. Note that $il(q) \leq L_q$, the maximum queue size. In some cases, we impose further constraints on the initial queue length to ensure consistency with the output count deq for q's port. For example, when $deq < T$, the number of time steps in the interval, and the output port has a single queue, the initial queue length must not exceed deq. Otherwise, the queue would dequeue more than deq packets, conflicting with the observed output count.

Lower bound on number of packets dequeued until t. We calculate the minimum number of packets that q will dequeue until t, assuming the query is true. For our example query $QgeK$ to be true, queue length at t must be at least $K > 0$. Depending on the value of K, the queue length must be non-zero for some minimum number of time steps $nz(K)$ before t, during which time it will dequeue at its minimum dequeue rate mdr. Hence, $cdeq(q,t) \geq nz(K) * mdr$. We calculate $nz(K)$ assuming the queue receives packets at the maximum possible rate (according to the input packet counts) for as long as possible until it reaches a queue length of K. We consider two (exhaustive) cases to determine the initial queue length (1) queue has a packet to send at every time step before t (queue length must become at least K at t starting from the maximum possible value of $il(q)$) and (2) queue has no packet to send at some time step before t (in this case, queue length must become at least K at t starting from a queue length of zero.)

Lower bound on number of packets dropped until t. We use $cdrp(q,t) \geq 0$. We use a tighter lower bound if the query specifies $st = end = T$ (which makes $t = T$) and queue q is the only queue in its port; then, $cdrp(q,T) = drp_i$, where drp_i is the dropped packet count for q's port.

Lemma 5.2 (CSG correctness)**.** *CSG returns a formula F_C representing a set C that contains all packet traces consistent with the query, output counts, and dropped counts,* i.e., $M_o \cap Q \subseteq C$.

Proof sketch. The set C represented by formula F_C is equivalent to $M'_o \cap Q'$ (by construction). M'_o is obtained from the packet conservation equation (4) assuming consistency with output and dropped counts, so any packet trace consistent with output and dropped counts will satisfy M'_o, *i.e.,* $M_o \subseteq M'_o$. For queries over *enq* or *cenq* metrics, Q' is obtained by eliminating quantifiers in the query Q to obtain an equivalent formula, so $Q' = Q$ in this case. For queries over *qlen*, Q' is derived starting from the packet conservation equation (5) (which is true for all packet traces) and enforcing bounds on terms assuming that the query is true. Hence, any trace that satisfies the query will also satisfy Q'. For all three metrics, $Q \subseteq Q'$. Since $M_o \subseteq M'_o$ and $Q \subseteq Q'$ it follows that $M_o \cap Q \subseteq M'_o \cap Q'$, so $M_o \cap Q \subseteq C$.

5.2 Computing Representative Abstract Traces

For each component F_C^j in the cover-set $F_C = F_C^1 \vee ... \vee F_C^k$ generated by the CSG, the MUC constructs a representative abstract trace R_j that satisfies F_C^j .

Representative is most-uniform. We characterize a representative abstract trace R_j satisfying F_C^j as being "most-uniform" among all abstract traces satisfying F_C^j. A most-uniform abstract trace has packets in columns of lowest possible *height* subject to satisfying F_C^j, where height of a column denotes the sum of its entries. Formally, for all $i \geq 1$, the total height of the i highest columns in R_j must be no more that the total height of the i highest columns in an arbitrary abstract trace A satisfying F_C^j.

Algorithm 2 (shown in Appendix A) takes as input F_C^j, a conjunction of constraints of three types: *LB* ($enq(q,t) \geq lb$), *CLB* ($cenq(q,t) \geq lb$), and *CUB* ($cenq(q,t) \leq ub$). It constructs an abstract trace R_j that is most-uniform by starting with the null matrix (with all entries 0) and incrementing appropriate matrix entries in order to satisfy the constraints. The procedure can be thought of as placing packets at appropriate time steps and queues in order to satisfy the given constraints. Importantly, the algorithm places each packet in a time step with lowest height (*i.e.,*, with minimum number of packets in total across all output queues) subject to upper bound constraints. This allocation strategy prioritizes placing packets in time steps that have few packets, making the resultant arrangement as uniform as possible subject to satisfying the given constraints.

Theorem 5.3 (Reduction via representative abstract trace)**.** *If the representative abstract trace R_j of a component F_C is not input-consistent, then every abstract trace A satisfying F_C^j is not input-consistent.*

Proof sketch. Consider an arbitrary A satisfying F_C^j. We show that if A is consistent with input counts then the output of Algorithm 2, R_j, must be consistent with input counts. Since A is consistent with input counts, by the definition of input-consistency, there exists some labeling function mapping each packet in A to an input port such that each input port i is used exactly $rcv[i]$ times (where rcv denotes the input packet counts) and packets at the same time step in A are mapped to different input ports. We show that is possible to transform A to R_j by shifting packets from one time step to another so that packets at the same time step continue to be mapped to different input ports after each shift. After performing all these shifts to reach R_j, the labels of packets in R_j will be a valid labeling; hence, R_j is consistent with input counts. The full proof is in Appendix B.

5.3 Checking Consistency with Input Counts

The Matrix-based Consistency Checker (MCC) checks if a given representative abstract trace R_j contains a packet trace that is consistent with input packet counts rcv. Recall that an abstract trace does not specify the input port each packet came from, and represents a set of packet traces, one packet trace for each labeling of packets with input ports.

There could be a huge number of possible labelings of packets in an abstract trace so it is impractical to iterate through all labelings to check if one is consistent with input counts. Constraint solving approaches will also have scalability limitations due to the large search space of labelings.

Reduction to $(0,1)$-matrix existence. We reduce the problem of determining if an abstract trace R_j contains a packet trace consistent with input counts to checking the existence of a $(0,1)$-matrix with given row sums and column sums. We leverage a known combinatorics result (the Gale-Ryser theorem [13, 24]) on matrix existence to solve it efficiently. The idea behind this reduction is that an abstract trace can be equivalently represented as a $(0,1)$-matrix with a row for each input port and a column for each time step. The given abstract trace R_j fixes the sum of each column and the input counts rcv fix the sum of each row of this $(0,1)$-matrix. R_j is consistent with input counts iff such a $(0,1)$-matrix exists for input counts rcv. We prove this in Appendix C.

Theorem 5.4 (Abstract Trace Consistency). *An abstract trace R_j ($N_q \times T$ matrix) is consistent with input counts rcv iff There exists an $N \times T$ $(0,1)$-matrix X s.t.*

$$\forall i.\ rsum(X,i) = rcv[i] \text{ and } \forall t.\ csum(X,t) = csum(R_j,t)$$

where $rsum(X, i)$ and $csum(X, t)$ denote the sum of entries in row i and column t respectively of matrix X.

Algorithm 1 checks input-consistency of an abstract trace using the Gale-Ryser theorem [13, 24]. The Gale-Ryser theorem [13, 24] states that there exists a $(0,1)$-matrix with column sums c and row sums r, where c and r are partitions of a positive integer, if and only if r is dominated by c', where c' is the conjugate partition [3] of c.

Algorithm 1: Matrix-based consistency check

```
Input  : $R_j$, abstract trace ($N_q \times T$ matrix)
         $rcv$, input counts ($1 \times N$ matrix)
Output : True iff $R_j$ has a packet trace satisfying $rcv$
Function check_consistency($R_j$, $rcv$):
    $rt : 1 \times T$ // Aggregate rate of $R_j$
    for $t = 0; t < T; t{+}{+}$ do
        $rt[t] \leftarrow \sum_{q \in N_q} R_j[q][t]$ // column sums
    $rt$.sort(); $rcv$.sort()// descending
    return matrix_exist($rcv$, $rt$)
Function matrix_exist($r$, $c$):
    $rsums[0] = r[0]$// cumulative row sums
    for $i = 0; i < r.size(); i{+}{+}$ do
        $rsums[i] = rsums[i-1] + r[i]$
    $c' \leftarrow conj(c)$// conjugate partition of $c$
    $csums[0] = c'[0]$ // cumulative sums of $c'$
    for $i = 0; i < c'.size(); i{+}{+}$ do
        $csums[i] = csums[i-1] + c'[i]$
    for $i = 0; i < r.size(); i{+}{+}$ do
        if $rsums[i] > csums[i]$ then
            return $False$ // $c$ doesn't dominate $r$
    return $True$
```

6 QUASI: Layer 2

QUASI-2 models switch operation at each time step using Satisfiability Modulo Theories (SMT) [5] constraints, with additional constraints enforcing the observed packet counts and the query. It uses the Z3 [11] solver to check if these constraints are satisfiable. If satisfiable, the query is true for some packet trace consistent with measurements; otherwise, the query can never be true for the given measurements.

Enqueue-rate abstraction enables N times fewer variables. QUASI-2 models a symbolic abstract trace which specifies the number of packets entering each output queue at each time step, requiring $O(N_qT)$ variables. In contrast, a symbolic packet trace (without enqueue-rate abstraction) specifies which input port sends a packet to which output queue at which time step and requires $O(NN_qT)$ variables.

Enqueue-rate abstraction is lossless for our query language, guaranteeing correctness of QUASI-2. Although the enqueue-rate abstraction ignores input ports of packets, it does not lose any information since our queries do not distinguish between packets in an output queue based on their input ports.

7 Evaluation

Our evaluation demonstrates QUASI 's usefulness by showing that QUASI can conclusively answer queries that cannot be answered by our baseline, *i.e.,* a conservative heuristic analysis of the measurements. We go further than answering yes/no queries, extracting quantitative bounds on metrics like queue size and buffer occupancy from per-port packet counts. We show that QUASI 's bounds are significantly tighter than the bounds inferred by our baseline, demonstrating that QUASI can infer useful information about metrics (like queue size) that are not directly measured. We study how QUASI scales with the number of time steps in a monitoring interval and compare it with an existing performance analysis tool, Fperf [4].

Our measurements admit interesting scenarios. We collect per-port packet counts using the ns-3 simulator [22] with a star topology consisting of a switch with 8 ports, with hosts sending randomized UDP traffic. Each switch output port can queue a maximum of 250 packets and the total buffer size is 2000 packets. We ensure our collected measurements allow the possibility of packets accumulating in queues, which leads to interesting performance scenarios like bursts and increased queueing latency. We evaluate QUASI on a set of 25 monitoring intervals of duration 100 time steps, each interval corresponding to a different set of measurements. We use *example* to refer to one monitoring interval. We use a smaller interval ($T = 10$) for comparison with FPerf, and several intervals of increasing duration to evaluate QUASI's scalability.

We select relevant, non-trivial queries. Our queries are drawn from real-world use cases: (1) BurstOccurrence ("Could a burst of rate R and duration D occur sometime during the interval?") is useful for troubleshooting poor application performance, (2) QlenK ("Could the queue length at port O_i be at least K?") is useful for checking latency SLO compliance, (3) MaxQlen ("What is the maximum queue length at port O_i during the interval?") is useful for estimating queueing latency, and (4) MaxBuff ("What is the maximum buffer occupancy during the interval?") is useful for buffer provisioning. Our queries involve multiple performance metrics (queue length and enqueue-rate) and the MaxBuff query involves multiple queues. We demonstrate that our selected queries cannot be easily answered by a heuristic analysis of the measurements, implemented in our HEURISTIC baseline: HEURISTIC fails to answer many queries that QuASI conclusively answers.

We use a strong baseline. As a baseline, we implement a conservative analysis, HEURISTIC, that answers a query negatively if it conflicts with a set of heuristics/rules that must always hold. HEURISTIC incorporates several non-trivial rules based on switch operation, including accounting for ports not included in the query. For burst queries, it uses rules: (1) number of packets in burst cannot exceed total number of packets received by the switch, (2) rate of incoming packets to an output queue cannot exceed the number of input ports that receive packets during the interval, and (3) a burst cannot occur in an output port if the minimum number of packets it dequeues is larger than the port's output count. Note that HEURISTIC will never report false negatives but could report false positives.

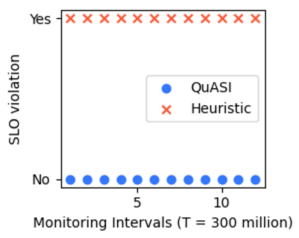


Figure 7: QUASI proves latency SLO compliance in each interval while HEURISTIC incorrectly reports an SLO violation.

7.1 Case Study: Checking SLO Compliance

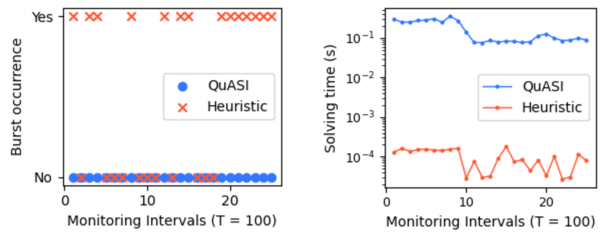
Consider, as in §3.2, a cloud operator who receives a complaint from a customer claiming the latency between the customer's servers, connected by a single switch, violated the Service Level Objective (SLO) of 289μs between 9 am and 10 am on a given day. The operator has SNMP [12] per-port packet counts over 5-minute intervals, which we collect via simulation using a switch with 8 ports, 8Gbps bandwidth, and no more than 2500 packets queued per output port. We generate randomized traffic using 1KB packets. Packet transmission time is 1μs, so a 5-minute interval has 300 million time steps.

To check for an SLO violation, we use the QlenK query which asks if a port could queue at least K packets, with K set to 290 (the SLO specifies no more than 289 queued packets). We run QUASI and HEURISTIC on each 5-minute interval in the specified hour. The results are shown in Fig. 7. HEURISTIC reports that the queue length was at least 290 packets during each interval, failing to prove SLO compliance. In comparison, QUASI reports that the queue length was always below 290 packets during each interval, thereby proving SLO compliance. QUASI-1 answers each query negatively so QUASI-2 is not run. QUASI took only 0.03s in total to prove SLO compliance, demonstrating that QUASI can scale to realistic (minute-granularity) monitoring intervals.

7.2 Burst Occurrence

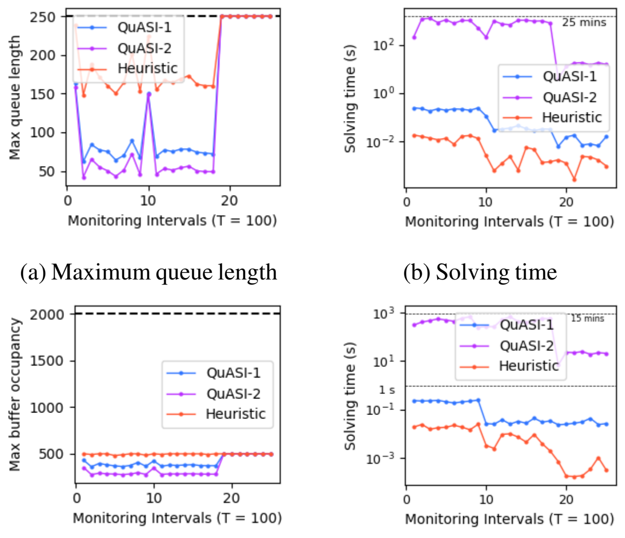
We evaluate QUASI on the BurstOccurrence query which asks if a burst of given rate and duration could occur in an interval. Fig. 8a shows the reported answer, with "yes" denoting a burst could have occurred and "no" denoting a burst is impossible. We have purposefully only included queries that could not have happened to see how often QUASI-1 would have a false positive and how long it would take for QUASI-2 to solve. Fig. 8b shows the solving time.

QUASI proves the absence of bursts in all intervals. QUASI-1 answers the query negatively for all examples, so we do not run QUASI-2 (since QUASI-1 will never report a false negative). HEURISTIC incorrectly reports that a burst is possible for 14 out of 25 examples, demonstrating that QUASI can conclusively negatively answer queries that a heuristic analysis cannot. Fig. 8b shows that QUASI is



(a) Burst occurrence (b) Solving time

Figure 8: QUASI proves the absence of bursts within 1s, while HEURISTIC incorrectly reports some bursts are possible.



(a) Maximum queue length (b) Solving time

(c) Maximum buffer occupancy (d) Solving time

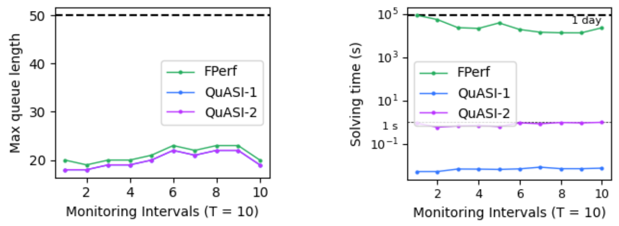
Figure 9: HEURISTIC results are consistently looser compared to QUASI. Even QUASI-1 (*i.e.,* the first layer alone) can provide tighter bounds within a fraction of a second.

efficient: QUASI-1 takes less than 1s for each example. Here we observe the benefit of a layered approach. Having a less accurate but much more scalable first layer removes the need for running the exact (but less scalable) analysis here.

7.3 Max Queue Length and Buffer Occupancy

We evaluate QUASI on queries MaxQlen and MaxBuff which ask the maximum queue length and buffer occupancy respectively during a monitoring interval.

Using QUASI to estimate quantitative bounds. We implement a binary search over the value of the metric (queue length/buffer occupancy), issuing a sequence of parameterized Boolean queries, each query asking if the value of the metric can be no less than a constant K. For MaxQlen, we use the Boolean query "$\exists t \in [1,T] \sum_{q \in O_i} qlen(q,t) \geq K$?, which can be read as "Can port O_i ever have at least K packets queued at some time during the monitoring interval?", where O_i is a specified port and K is a parameter. We initialize K to L_i, the maximum queue size at port O_i. The answer reported by QUASI (yes/no) determines how K is varied (increased or decreased) in the next Boolean query. If QUASI says "No", it means the queue length is always below K during the interval. Hence, we decrease the value of K in the next query. This process continues until the binary search terminates (QUASI zeroes in on a single value of K) or the minimum/maximum limit of the metric is reached. The final value of K is returned as QUASI's estimate of the maximum possible value of the metric for the given packet counts.



(a) Maximum queue length (b) Solving time

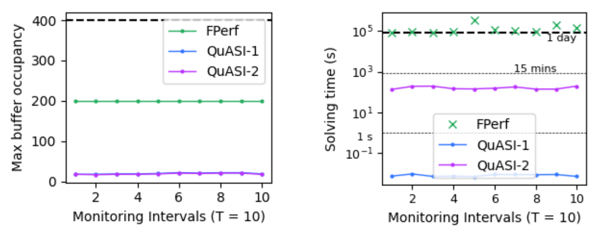
Figure 10: QUASI is up to 10^6x faster than FPerf for MaxQlen.

As QUASI-1 will never report a false negative, but could report a false positive, it over-estimates the value so that the final answer is an upper bound on the true maximum value. QUASI-2, which is an exact analysis, will report the correct answer to each Boolean query, so its final answer will be the true maximum value. We use the answer returned by QUASI-2 as ground truth to evaluate the accuracy of QUASI-1.

Fig. 9 shows the reported estimates of maximum queue length and buffer occupancy (lower is better as the estimates are upper bounds) and solving time for 25 monitoring intervals (each interval is a new set of measurements). We independently evaluate QUASI's two layers to show their difference in running time and the effect of false positives in QUASI-1.

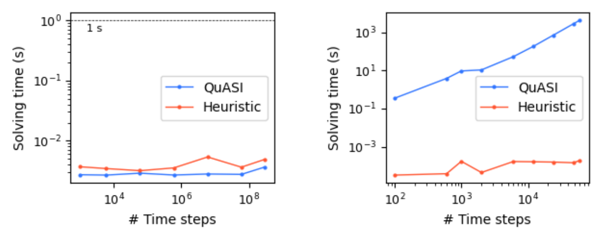
QUASI finds upper bounds within a second and the exact answer within 25 **minutes for both metrics.** Thanks to its layered approach, QUASI quickly finds upper bounds for both metrics (with avg. relative error of 0.25) within a second for each interval. QUASI finds the true maximum value over each interval within 25 minutes for queue length, and within 15 minutes for buffer occupancy.

QUASI-1 finds bounds up to 58% **tighter than bounds found by HEURISTIC.** Figs. 9a and 9c show that for both metrics, QUASI-1 reports bounds (shown in blue) that are consistently lower than bounds found by HEURISTIC (shown in orange), demonstrating that QUASI's approximate layer provides valuable insights that cannot be obtained by a heuristic analysis of the measurements. Our implementation of HEURISTIC is fair: its estimates are significantly below the trivial upper bounds of maximum queue size (250) and total buffer size (2000) shown by black dashed lines in Figs. 9a and 9c.



(a) Maximum buffer occupancy (b) Solving time

Figure 11: FPerf fails to compute the maximum buffer occupancy within a day while QUASI takes less than 15 minutes.



(a) QlenK (b) BurstOccurrence

Figure 12: QUASI takes almost constant time for QlenK and scales quadratically with interval size for BurstOccurrence.

7.4 Comparison with FPerf

To evaluate FPerf on our problem, we disable its workload synthesis step and stop after FPerf checks if some trace consistent with the base workload satisfies the query. We specify a base workload that enforces input packet counts (FPerf does not allow constraints over dequeued packets, so we do not use output and dropped packet counts). We extend FPerf to answer our quantitative queries MaxQlen and MaxBuff. FPerf does not support our BurstOccurrence query. We reduce the monitoring interval to just 10 time steps (from 100 steps in preceding subsections) so that Fperf terminates sometimes. We report FPerf's estimates after running it up to ≈ 1 day for each interval.

QUASI computes the maximum queue length 10^6X faster than FPerf. FPerf takes an average of 8.5 hours to find the maximum queue length while QUASI takes less than a second. The answers reported by FPerf and QUASI match when FPerf terminates within a day (*i.e.,* for 9 out of 10 intervals); the difference of 1 in Fig. 10a is because FPerf calculates queue length before dequeueing a packet rather than after.

FPerf does not complete all per-interval required queries in 1 day, reporting an upper bound ≈ 9 times larger than the true maximum buffer occupancy; QUASI finds the exact answer in under 15 minutes. We left FPerf running for more than 2 days for one example but that did not improve its estimate. As shown by the green curve in Fig. 11a, FPerf finds a very loose upper bound; the purple curve shows the exact answer found by QUASI-2. QUASI-1 is 100% accurate on all these examples, reporting the same answer as QUASI-2 (which we regard as ground truth).

7.5 Scalability

QUASI 's verification time increases as the number of time steps in the monitoring interval increases. This is because a larger number of time steps leads to larger abstract trace matrix dimensions in QUASI-1 and more variables in QUASI-2's SMT encoding. Increased switch bandwidth (which decreases packet transmission time) and longer monitoring intervals both lead to more time steps in the monitoring interval. We evaluate QUASI's scalability with the number of time steps in the monitoring interval for two queries: QlenK and BurstOccurrence; and report solving time in Fig. 12. QUASI-1 answers all queries negatively, so QUASI-2 does not run.

QUASI's solving time scales quadratically with number of time steps for BurstOccurrence. For the BurstOccurrence query, QUASI generates a cover-set with $O(T)$ components, and takes about 75 minutes for 60,000 time steps. For QlenK, QUASI's solving time remains about the same as the monitoring interval size is increased, answering queries for intervals with millions of time steps within 1 sec. This is because CSG detects that the generated cover-set does not contain any packet traces, so subsequent components in QUASI-1 are skipped. These queries are not trivial: HEURISTIC incorrectly answers all of them positively.

QUASI scales to realistic monitoring intervals (5 mins) for queue length queries. QUASI answers the queue length query negatively within a second for an interval with 300 million time steps (*i.e.,* 5 minutes, as packet transmission time is 1μs). HEURISTIC answers the query positively, showing again that QUASI can quickly answer non-trivial queries.

8 Conclusion

This paper presents QUASI, a layered approach to answering queue-related queries using coarse-grained packet counts. QUASI is centered around the novel and lossless enqueue-rate abstraction. The first layer of QUASI performs an over-approximate analysis using representations of packet traces as (0,1)-matrices. The second layer utilizes an exact but less scalable analysis based on use of an SMT solver. QUASI is 10^6X faster than state-of-the-art while answering non-trivial queries about queue metrics.

Acknowledgments

We would like to thank the NSDI reviewers and our shepherd for their comments. We are grateful to Prof. Matt Weinberg for helpful feedback in the early stages of this work. This work was supported in part by NSF Grants CNS-2319442 and CNS-2312539.

References

[1] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. ABM: Active Buffer Management in Datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022.

[2] Vamsi Addanki, Wei Bai, Stefan Schmid, and Maria Apostolaki. Reverie: Low pass {Filter-Based} switch buffer sharing for datacenters with {RDMA} and {TCP} traffic. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 651–668, 2024.

[3] George E Andrews. *The theory of partitions*. Cambridge university press, 1998.

[4] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. Formal Methods for Network Performance Analysis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 645–661, 2023.

[5] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[6] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. SIGCOMM '20, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[8] Armin Biere and Daniel Kröning. Sat-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer, 2018.

[9] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 15–29, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[12] Mark Fedor, Martin Lee Schoffstall, James R. Davin, and Dr. Jeff D. Case. Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[13] David Gale. A theorem on flows in networks. In *Classic Papers in Combinatorics*, pages 259–268. Springer, 1956.

[14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 549–564, 2019.

[15] Fengchen Gong, Divya Raghunathan, Aarti Gupta, and Maria Apostolaki. Zoom2Net: Constrained Network Telemetry Imputation. In *Proceedings of the ACM SIGCOMM 2024*, 2024.

[16] In-Band Network Telemetry A Powerful Analytics Framework for your Data Center.

[17] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In *Handbook of Model Checking*, pages 447–491. Springer, 2018.

[18] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–8, 2018.

[19] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 202–208, 2009.

[20] Yiran Lei, Liangcheng Yu, Vincent Liu, and Mingwei Xu. Printqueue: performance diagnosis via queue measurement in the data plane. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 516–529, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Nick McKeown, Guido Appenzeller, and Isaac Keslassy. Sizing router buffers (reduxß). *ACM SIGCOMM Computer Communication Review*, Oct. 2019.

[22] Ns3 network simulator. https://www.nsnam.org/.

[23] Paula Roquero and Javier Aracil. On performance and scalability of cost-effective snmp managers for large-scale polling. *IEEE Access*, 9:7374–7383, 2021.

[24] H. J. Ryser. Combinatorial properties of matrices of zeros and ones. *Canadian Journal of Mathematics*, 9:371–377, 1957.

[25] Jurgen Schonwalder, Aiko Pras, Matus Harvan, Jorrit Schippers, and Remco van de Meent. Snmp traffic analysis: approaches, tools, and first results. In *2007 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 323–332. IEEE, 2007.

[26] Yang Zhou, Ying Zhang, Minlan Yu, Guangyu Wang, Dexter Cao, Eric Sung, and Starsky Wong. Evolvable network telemetry at facebook. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 961–975, 2022.

A MUC Algorithm

The MUC uses Algorithm 2 to construct a most-uniform abstract trace for a given cover-set component.

Algorithm 2 takes as input a cover-set component, consisting of three types of constraints, and the total number of packets received in the interval (P). It then constructs a representative abstract trace starting from an empty abstract trace (with no packets) and iteratively placing packets to be enqueued at appropriate output queues at specific time steps in order to satisfy the given constraints.

First, lower bound (LB) constraints are satisfied (lines 2 to 4); the **get_lb** function returns the lower bound lb given queue q and time step t. The algorithm sets $R[q][t]$ to lb; this is the minimum number of packets needed to satisfy this constraint.

Second, cumulative lower bound (CLB) constraints are satisfied (lines 6 to 11). Note that there are many ways to distribute packets in an interval $[1,t]$ to satisfy such a constraint of the form $cenq(q,t) \geq lb$. Algorithm 2 uses the **fill** function, which places packets until the given lower bound constraint is satisfied, or until upper bound constraints prevent filling additional packets. Each packet is placed at time step $place \in ts$ which has the minimum number of packets in total (across output queues); ts is the set of time steps in $[1,t]$ that can take at least one packet without violating upper bounds. This allocation strategy prioritizes placing packets in time steps that have few packets, making the resultant arrangement as uniform as possible subject to satisfying the given constraints.

If the **fill** function returns abstract trace R that does not respect the lower bound lb, it means the upper bound constraints did not allow placing any more packets. In this case, there is no way to satisfy the lower bound constraint while respecting the upper bound constraints, so allocation fails and the algorithm returns *False*.

Algorithm 2: MUC

```
Input  : LB, lower bounds at given time steps
         CLB, lower bounds until given time steps
         CUB, upper bounds until given time steps
         $P$, total number of packets received
Output : $R_j$, most-uniform abstract trace
         for given constraints; False if it does not exist
1  int $R[N_q][T]$ // Empty abstract trace
2  for $t,q \in LB$ do
3      $lb \leftarrow$ get_lb$(LB,t,q)$
4      $R[q][t] \leftarrow lb$
5  $time\_pts \leftarrow$ get_sorted_time_pts$(CUB,CLB)$
6  for $t$ in $time\_pts$ do
7      for $q \in [1,N_q]$ do
8          $lb \leftarrow$ get_lb$(CLB,t,q)$
9          $R \leftarrow$ fill$(R,t,q,lb,CUB,N)$
10         if tot$(R,t,q) < lb$ then
11             return False
12 $R \leftarrow$ fill$(R,T,any_q,P,CUB,N)$
13 if tot$(R,T) > P$ then
14     return False
15 return $R$
16 Function fill ($R$, $end$, $q$, $lb$, $CUB$, $N$):
       // Ensure $\sum_{t=1}^{end} R[q][t] \geq lb$
17     while tot$(R,end,q) < lb$ do
18         $ts \leftarrow$ ts_with_space$(R,end,CUB)$
19         if $|ts| = 0$ then
20             break
21         $place \leftarrow$ minHeightIndex$(ts)$
22         $R[q][place] \leftarrow R[q][place]+1$
23     return $R$
```

After satisfying CLB constraints, the algorithm fills any remaining packets to make the total number of packets in R equal to P (line 12); the argument any_q indicates the packets can be filled in any queue subject to satisfying the upper bound (CUB) constraints. If the total number of packets exceeds P, there are insufficient packets to satisfy the constraints and the algorithm returns *False* (line 14). Otherwise, R is returned.

If Algorithm 2 returns an abstract trace, it is most-uniform. The algorithm ensures that its output R satisfies the given constraints (*i.e.,* the cover-set component). The **fill** function always places a packet in a time step with the fewest possible packets subject to satisfying the constraints, hence ensuring the output R is most-uniform. The algorithm does not return an abstract trace if the given cover-set component is unsatisfiable, *i.e.,* if the lower bound constraints cannot be satisfied along with the upper bound constraints or if the number of packets received P is insufficient to satisfy the given constraints.

B MUC Correctness

We present the proof of Theorem 5.3, which states that if the representative abstract trace R_j of a component F_C is not input-consistent, then every abstract trace A satisfying F_C^j is not input-consistent.

We begin by defining some notation. For an abstract trace A, $csum(A, t_1)$ denotes the sum of entries in column t_1 of A. We will also use the term "height of t" to mean the sum of entries at column t. We use R_j to denote the most-uniform abstract trace returned by Algorithm 2 for constraints F_C^j. For an abstract trace A, $ht_A \in \mathbb{N}^T$ denotes the row vector of column heights (where height of a column is the sum of its entries), sorted in non-increasing order, *i.e.,* $ht_A[1] \geq ... \geq ht_A[T]$. We use ht_{R_j} to denote the vector of sorted column heights for R_j. We use $v[:i]$ to denote the vector of the first i entries of v; and $\sum v[:i]$ to denote the sum of the first i entries in v.

Next, we define a transformation of an abstract trace, called a balancing shift, which preserves input-consistency.

Definition B.1 (Balancing shift)**.** *For an abstract trace A, a balancing shift moves a packet from time step t_1 to time step t_2 where $csum(A, t_1) > csum(A, t_2)$.*

A balancing shift moves a packet from a column of higher height to a column of lower height. A balancing shift from t_1 to t_2 requires that the height of t_1 is larger than the height of t_2, and will never increase the height difference between t_1 and t_2.

Lemma B.2 (Balancing shifts preserve input-consistency)**.** *For any abstract trace A, let A' be the abstract trace obtained by performing a balancing shift on A. If A is input-consistent, then A' must also be input-consistent.*

Proof. Assume that A is input-consistent. Then there is a labeling Lbl that associates each packet in A with an input port such that packets at the same time step in A are mapped to distinct input ports. Let the balancing shift be from t_1 to t_2 where $csum(A, t_1) > csum(A, t_2)$. Since column t_1 has at least one more packet than column t_2 and packets at the same time step are mapped to different input ports, it is always possible to choose a packet p at t_1 that has an input port distinct from all packets at t_2. (If this were not the case, it would mean that at least two packets at t_1 were mapped to the same input port, which is a contradiction). Shift packet p from t_1 to t_2 to obtain abstract trace A'. The balancing shift did not change the number of input port labels, so A' will have each input port label appearing as many times as the port's input count. Moreover, A' has packets at each time step mapped to distinct input ports by Lbl, so A' is input-consistent. □

We now show that the abstract trace R_j returned by Algorithm 2 is the "most-uniform" in C_j, the set of abstract traces satisfying a cover-set component F_C^j.

Lemma B.3. *The abstract trace R_j returned by Algorithm 2 is the most-uniform in C_j,* i.e., *for any abstract trace $A \in C_j$, the total height of the i largest columns in R_j must be no more than the total height of the i largest columns in A, for any $i \geq 1$.*

$$\forall A \in C_j.\ \forall i \geq 1.\ \sum ht_{R_j}[:i] \leq \sum ht_A[:i] \tag{6}$$

Proof. The proof is by contradiction. Assume R_j is not most-uniform (*i.e.,* eqn. (6) does not hold). Then, there exists some $A \in C_j$ and some $i \in [1, T]$ such that

$$\begin{aligned} &\sum ht_{R_j}[:i] > \sum ht_A[:i] \text{ , and} \\ \forall k < i.\ &\sum ht_{R_j}[:k] \leq \sum ht_A[:k] \end{aligned} \tag{7}$$

Note that $ht_{R_j}[i] > ht_A[i]$; otherwise, $\sum ht_{R_j}[:i]$ could not be greater than $\sum ht_A[:i]$. Let h denote the value of $ht_A[i]$. Consider the total number of packets that A and R_j pack in columns of height no more than h.

Let P denote the total number of packets in A and R_j.

$$P = \sum ht_A = \sum ht_{R_j} \tag{8}$$

Then, A packs more than $P - \sum ht_A[:i]$ packets in columns of height $\leq h$. Since $ht_{R_j}[i] > ht_A[i]$ and ht_{R_j} is sorted in non-increasing order, R_j packs no more than $P - \sum ht_{R_j}[:i]$ packets in columns of height $\leq h$. From eqn. (7), $\sum ht_{R_j}[:i] > \sum ht_A[:i]$, so R_j packs fewer packets than A in columns of height $\leq h$. Algorithm 2 constructs R_j by placing each packet at the minimum possible height subject to satisfying the given constraints. This allocation strategy packs as many packets as possible in columns of low height. Since A satisfies the constraints F_C^j, the total number of packets in columns of height $\leq h$ in A must be no more than the corresponding number of packets in R_j. Assuming there exists an $A \in C_j$ such that R_j is less uniform than A (eqn. (7)) implies that A packs strictly more packets than R_j in columns of height $\leq h$, which is a contradiction. □

We now present the proof of Theorem 5.3. We show that if some $A \in C_j$ is input-consistent, then the most-uniform abstract trace R_j must also be input-consistent. We define a procedure `Relabel` that transforms the labeling function f_L for A to construct a valid labeling function for R_j, thereby showing that R_j is input-consistent.

Proof. Assume there is some $A \in C_j$ which is input-consistent. Let f_L denote the labeling function which assigns each packet in A to an integer denoting an input port. Function f_L satisfies two conditions: (1) number of input port labels is equal to the port's input count; and (2) packets at the same column in A are mapped to distinct input port labels.

We define a procedure `Relabel` which transforms the labeling function f_L into a valid labeling function f'_L for the most-uniform abstract trace R_j. `Relabel` operates on the sorted height vectors ht_A and ht_{R_j} for A and R_j respectively.

`Relabel` consists of T steps. In step i, `Relabel` sets entry i of ht_A equal to entry i of ht_{R_j}; we use $ht'_A[i]$ to denote the value of $ht_A[i]$ after modification. At each step, `Relabel` stores the number of extra packets $excess[i] = \max(0, ht_A[i] - ht_{R_j}[i])$. If $ht_A[i] < ht_{R_j}[i]$ it uses the extra packets so far to make $ht_A[i]$ equal to $ht_{R_j}[i]$ (we will show that there will always be enough extra packets). After T steps, each entry of ht'_A will be equal to the corresponding entry of ht_{R_j}, *i.e.,* $ht'_A = ht_{R_j}$ at the end of the procedure.

We will now show that `Relabel` transforms ht_A into ht_{R_j} by only using balancing shifts.

By Lemma B.3, $\sum ht_{R_j}[:i] \leq \sum ht_A[:i]$ for all $i \in [1,T]$. In step 1, we will not have a deficit of packets since $ht_{R_j}[1] \leq ht_A[1]$.

Consider any step $i \geq 2$ in which there is a shortage of packets in ht_A, *i.e.,* where $ht_A[i] < ht_{R_j}[i]$. Since $\sum ht_A[:i] \geq \sum ht_{R_j}[:i]$ and $\sum ht_A[:i-1] \geq \sum ht_{R_j}[:i-1]$, it is guaranteed that the total excess packets so far, $ht_A[:i-1] - \sum ht_{R_j}[:i-1]$, is no less than the deficit at i, $ht_{R_j}[i] - ht_A[i]$.

$$\sum ht_A[:i] \geq ht_{R_j}[:i] \tag{9}$$

$$\sum ht_A[:i-1] + ht_A[i] \geq ht_{R_j}[:i-1] + ht_{R_j}[i] \tag{10}$$

$$\sum ht_A[:i-1] - \sum ht_{R_j}[:i-1] \geq ht_{R_j}[i] - ht_A[i] \tag{11}$$

Hence, we will be able to make entry $ht_A[i]$ equal to $ht_{R_j}[i]$ by using some of these extra packets.

Any extra packet is from some entry $t < i$ where $ht_A[t] > ht_{R_j}[t]$ originally (*i.e.,* before `Relabel` changes ht_A). Given that there is a deficit at entry i in A and both ht_A and ht_{R_j} are sorted in non-increasing order,

$$ht_A[t] > ht_{R_j}[t] \geq ht_{R_j}[i] > ht_A[i] \tag{12}$$

which implies that $ht_A[t] > ht_A[i]$. Hence, shifting an extra packet to i is always from a column with greater height to a column of lower height, *i.e.,* it is always a balancing shift.

Since the only moves that `Relabel` uses are balancing shifts, by Lemma B.2, it is always possible to choose an extra packet to shift from entry t to entry i such that entry i will have each packet mapped to a distinct input port. After `Relabel` terminates, the resultant vector ht'_A will have each entry i mapped to $ht'_A[i]$ distinct input port labels. Since a balancing shift does not change any input port label (it just moves labeled packets), ht'_A will be labeled with each input port a number of times equal to the port's input count. Since $ht'_A = ht_{R_j}$, each entry in ht'_A corresponds to some column in matrix R_j. Using the labeling over ht'_A, associating each entry with its corresponding column in R_j, we obtain a labeling that associates each packet in abstract trace R_j with an input port label satisfying the requirements of a valid labeling function. Hence, R_j is input-consistent. □

C MCC Correctness

We present the proof of Theorem 5.4, which reduces the problem of checking input-consistency of an abstract trace to checking the existence of a $(0,1)$-matrix with given row sums and column sums.

Proof. $\Rightarrow$: Assume abstract trace R_j is consistent with input counts rcv. Then, there exists some labeling Lbl that associates each packet in R_j with an input port. We construct a $N \times T$ $(0,1)$-matrix X from R_j and Lbl with each entry $X[i][t]$ defined as follows:

$$X[i][t] = \begin{cases} 1 & \text{if } Lbl \text{ uses port } i \text{ at } t \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

Since Lbl uses each port i exactly $rcv[i]$ times, X will have exactly $rcv[i]$ 1s in row i, *i.e.,* the sum of row i will be $rcv[i]$ for all $i \in [1,N]$. Since Lbl labels all packets at the same time step in R_j with different ports, for each $t \in [1,T]$ the t^{th} column of X will have $csum(R_j, t)$ 1s, *i.e.,* $csum(X, t) = csum(R_j, t)$. Hence, R_j being input-consistent implies there exists a $(0,1)$-matrix X with its row sums determined by rcv and its column sums determined by R_j.

$\Leftarrow$: Assume there exists a $(0,1)$-matrix X with its row sums equal to the input counts rcv and its column sums equal to the column sums of abstract trace R_j. We construct a function Lbl from X. Let $AvlPorts(t)$ denote the set of indices of rows which have 1 in the t^{th} column of X. For each $t \in [1,T]$, let Lbl pick any $1-1$ mapping from $AvlPorts(t)$ to packets in the t^{th} column of R_j. Since $csum(X,\ t) = csum(R_j,\ t)$, $|AvlPorts(t)| = csum(R_j,\ t)$ so each packet enqueued at t can be labeled with a distinct input port. Since Lbl only uses ports from $AvlPorts(t)$ in each time step t (*i.e.,* only ports corresponding to entries of 1 in X for that time step), Lbl will use port i the number of times row i has entry 1 in X. Since $rsum(X,\ i) = rcv[i]$ for all $i \in [1,N]$ Lbl will use each port i exactly $rcv[i]$ times. Since Lbl satisfies the required properties in the definition of input-consistency and labels each packet in R_j, R_j is consistent with input counts rcv. □