IEEE Access
Multidisciplinary ⁞ Rapid Review ⁞ Open Access Journal

Received 13 September 2024, accepted 8 November 2024, date of publication 15 November 2024, date of current version 25 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3498737

RESEARCH ARTICLE

Tempus: Probabilistic Network Latency Verification

SEPEHR ABDOUS[1], **SENAPATI DIWANGKARA**[1], **AND SOUDEH GHORBANI**[1,2]
[1]Computer Science Department, Johns Hopkins University, Baltimore, MD 21218, USA
[2]Meta, Menlo Park, CA 94025, USA

Corresponding authors: Sepehr Abdous (sabdous1@jh.edu) and Senapati Diwangkara (diwangs@cs.jhu.edu)

This work was supported by the NSF Division of Computer and Network Systems (CNS) under Grant 1910821.

ABSTRACT Networks have exceedingly low latency requirements. Verifying network latency is crucial for identifying any bottlenecks that may negatively impact user experience and swiftness of business operations. Network operators today heavily rely on high-fidelity simulators to validate latency requirements. Alas, detailed simulators are slow and subsequently not scalable. Alternatively, network verifiers are emerging as powerful validation means. Network verifiers provide an abstract model of the network behavior. Albeit faster than their current simulation-based counterparts, abstracting the details of networks comes at a cost: the state-of-the-art verifiers have major limitations such as not modeling failures or latency that prevent them from reliably verifying latency. This paper bridges this gap by proposing a scalable latency verification method, Tempus, that decomposes latency verification into two phases (functional and temporal verification) and refines advanced abstract network models to enable fast temporal verification. Concretely, given a source and destination pair and the empirical latency measurements of network components (*e.g.,* the queueing delay), Tempus returns the probability of reaching the destination from the source within a time frame under all failure scenarios. We evaluate Tempus under both wide area and datacenter networks and show that it is fast and scalable. For instance, Parsimon, a state-of-the-art fast network simulator, requires more than one month to simulate all failure scenarios of an 8-ary fat-tree network with 100 Gbps links under 25% load. Tempus, in contrast, verifies the latency of the same network among all (*source*, *destination*) pairs and under all failure scenarios in only 8 minutes and 32 seconds, a speedup of three orders of magnitude. We also demonstrate that Tempus accurately approximates network latency under various degrees of load.

INDEX TERMS Delay estimation, system verification, performance evaluation, system performance.

I. INTRODUCTION

Catering to modern applications' increasingly low latency requirements pushes network operators to optimize every aspect of their network and set tight latency goals in their Service Level Agreements (SLAs) [1], [2], [3], [4], [5], [6], [7], [8]. For instance, Verizon's, AT&T's, and COMCAST's SLAs necessitate average Round-Trip Times (RTTs) of 45, 40, and 55 milliseconds, respectively [1], [2], and [3]. Verifying that the network complies with these increasingly strict latency requirements is crucial [9], [10], [11], [12]. High-fidelity packet-level simulators (*e.g.,* OMNeT++ [13], NS-3 [14], and DONS [15]) have emerged as the de facto latency estimation tools and are extensively used for testing various latency properties such as RTTs and flow completion times [7], [8], [16], [17], [18], [19], [20]. To ensure performance fidelity, these simulators try to accurately mimic fine-grained details of the network. This improves accuracy but comes with a great computational cost: The state-of-the-art network simulators are quite slow for verifying latency under numerous network states. For instance, using DONS [15], a recent parallel packet-level simulator optimized for speed, simulating a one-second run of a single state 8-ary fat-tree network (one with a single set of link failures) with 100 Gbps links under 25% load on a machine with 12 logical processors running up to 3.5 GHz

The associate editor coordinating the review of this manuscript and approving it for publication was Byung-Gyu Kim.

takes approximately 3 days.[1] Since an 8-arry fat-tree has 256 links, simulating *all* 2^{256} possible link failure scenarios would require more than 9.6×10^{74} years!

Network verifiers [4], [11], [12], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43] provide an alternative approach that strives to test *all* reachable states of an abstract model of the network compared to a few simulated ones. Alas, existing proposals on network latency verification have some key limitations: (i) They cannot model and verify latency under all failure scenarios [12], [44], [45], (ii) they are not general and only verify limited sources of delay such as network functions [10], [38], [46], smart NICs [47], or queueing delay [11], and (iii) they are coarse-grained and only support flow-level latency analysis [12], [39] which is not sufficient for increasing common packet-level round-trip time SLAs [1], [2], [3].

To address this gap, we introduce Tempus, a general and packet-level probabilistic temporal verification framework designed to verify the latency of packets as they traverse the network under various load balancing paradigms and across all failure scenarios. Given the network graph, the source and destination node, and a time threshold (latency bound), Tempus returns the *time-bounded reachability probability*, *i.e.,* the probability of the destination node being reachable from the source within the time threshold under all link failure scenarios. Tempus decomposes the latency verification task into two phases: (i) *functional verification*, in which Tempus computes the probability of two nodes being reachable under various network states, and (ii) *temporal verification*, in which Tempus calculates the probability of reaching a node from another one below the user-specified latency bound.

In the functional verification phase, to obtain the functional probability, Tempus refines the mathematical model of NetDice [23], a state-of-the-art qualitative verification framework, to explore all failure scenarios at scale. NetDice's key insight is identifying *hot edges*, *i.e.,* links in the network whose failure lead to changes in the paths between two nodes chosen by the routing protocol and load balancer. Recursively exploring reachable states by failing only hot edges enables NetDice to scale without compromising accuracy. The key distinction between Tempus's first phase and NetDice is that Tempus refines NetDice's model by augmenting every state with information about potential paths between the source and destination node, given the failed links. This information is crucial for calculating time-bounded reachability probability in the second phase.

After computing each state's functional probability, Tempus enters the second phase, temporal verification. It iteratively computes the temporal properties of each state by mapping delay distributions to every link's empirical latency measurements such as queueing, link propagation, and link transmission delays, obtained from measuring real networks, *e.g.,* [8], [48] and [49]. Tempus then applies numerical convolution [50] on these distributions to obtain the latency distribution of the hot edges and convolves the edge latency distributions, using the path information added to every network state in the previous phase, to obtain the potential paths' end-to-end latency distributions. After acquiring the latency distribution of each potential path, Tempus leverages its Cumulative Distribution Function (CDF) and the given time threshold to compute the time-bounded reachability probability of the path and averages over the path probabilities to calculate the network state's temporal probability. The final output is then generated by multiplying every state's temporal probability by its functional probability and summing over the results for all network states.

In addition to enabling latency verification, Tempus's second contribution is novel caching mechanisms that speed up the verification process. We observe that the set of potential paths for various network states is not mutually exclusive, *i.e.,* some paths can be shared between multiple states. We harness this property to optimize Tempus's verification time. We store the latency of each path and the latency corresponding to each set of paths and use the stored values instead of recalculating a path's or a set's latency in case they were observed again during the exploration. In §IV, we show that these optimizations considerably improve Tempus's temporal verification time, *e.g.,* these optimizations reduce Tempus's verification time by $316\times$ in an 8-ary fat-tree. Additionally, we illustrate that while caching information imposes some memory overhead, it reduces the RAM overhead of state-keeping by preventing repetitive computations. For instance, in an 8-ary fat-tree, while our caching mechanisms store $\sim$ 8 KB of data in the memory, they reduce the number of required convolutions by $87\times$ and result in a 21% reduction in Tempus's overall RAM consumption.

We implement Tempus (in Julia [51]) and evaluate it under various scales of datacenter and wide area networks. We observe that Tempus is scalable and provides a short verification time compared to high-fidelity simulation tools, *e.g.,* Tempus's verification time for a 16-ary fat-tree topology is $935,179\times$ and $103,206\times$ smaller compared to the time required by DONS [15] and Parsimon [52], respectively, for simulating the network states corresponding to all failure scenarios. We also evaluate Tempus's accuracy by comparing its estimation of the 99^{th} percentile packet-level latency to the tail end-to-end latency we observe from simulating a two-tier leaf-spine topology with 2 spines, 3 leafs, and 10 servers connected to each leaf. Our findings show that Tempus's approximation is precise irrespective of the network load. Specifically, as we increase the degree of load from 10% to 70%, Tempus accurately obtains the upper bound on the tail latency. In the rest of the paper, we explain the details of Tempus's design (§II) and optimizations (§III), describe our experiments and their results (§IV), and outline the

[1]Note that this is a significant speedup compared to pervasive simulators such as NS3 [14] and OMNeT [13]. Simulating the same scenario with OMNeT, for example, requires almost a week.

future directions (§V). We also discuss the related work and highlight the limitations of existing latency verifiers (§VI).

II. TEMPUS: PACKET-LEVEL LATENCY VERIFICATION

We present the design of *Tempus*, a framework that verifies *network latency*, *i.e.,* the latency imposed to packets while traversing through various components at the core of the network, under random link failures. In particular, given the source and destination nodes, the network topology graph, the components' empirical delay measurements, *e.g.,* the queueing delay, and a time threshold (latency bound), Tempus returns the time-bounded reachability probability, *i.e.,* the probability of reaching from the source node to the destination node below the time threshold, under all failure scenarios.

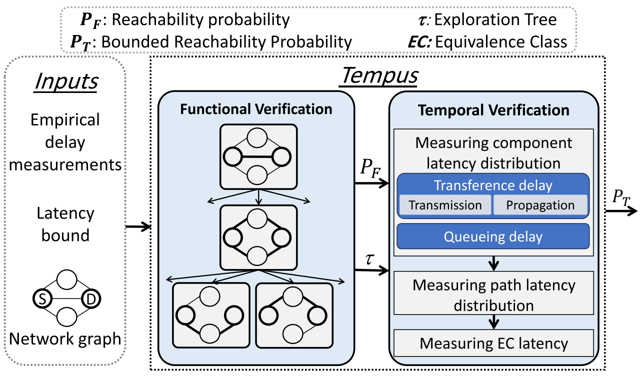

FIGURE 1. Tempus's architecture.

Tempus breaks the verification task into two phases: 1) *functional verification* phase, *i.e.,* verifying the reachability between two nodes under all link failure conditions, and 2) the *temporal verification* phase, *i.e.,* verifying that two nodes are reachable below the latency bound. Note that switch and router failures can be modeled as grouped link failures in which the group of links corresponding to the failed node are simultaneously considered down. We further discuss this in §V.

Figure 1 illustrates the general workflow of Tempus. During the functional verification phase (§II-A), given the network graph and a source and destination node, Ⓢ and Ⓓ, respectively, Tempus computes the probability that the destination is reachable from the source under various link failure scenarios. Specifically, given the routing protocol and the load balancing scheme which indicate the potential paths between source and destination, Tempus uses the network graph to obtain the exploration tree (τ), *i.e.,* a tree structure that represents the state of the network with regards to the potential paths between the source and the destination under all possible link failure scenarios, and uses the exploration tree to compute the reachability probability (P_F) between Ⓢ and Ⓓ in each state.

In the temporal verification phase (§II-B), Tempus incorporates time into the verification process by mapping a latency distribution to every link on the potential paths from source to destination. To this end, Tempus uses existing empirical latency measurements, *e.g.,* those obtained from measuring production networks [8], [48], [49], to map distributions to every link's queueing, propagation, and transmission delays and applies numerical convolution [50] on them to obtain the link's overall latency distribution. Next, Tempus exploits the link delay distributions to obtain the latency distribution of every potential path between the source and destination nodes. Lastly, using the path latency distributions, Tempus computes the probability of *time-bounded reachability* (P_T) between source and destination, *i.e.,* Ⓓ being reachable from Ⓢ under the input latency bound. In this work, we focus on two main sources of delay at the network core, queueing delay and the link transference delay, *i.e.,* the sum of the link's propagation and transmission delay. However, as we discuss in V, Tempus's method is general and can be extended to cover various sources of latency on packets' paths such as the extra latency imposed by network functions like firewalls or different layers of the hosts' networking stack. Next, we will discuss the details of Tempus's functional and temporal verification phases.

A. FUNCTIONAL VERIFICATION

To verify the functional property, inspired by NetDice [23], *we probabilistically verify the reachability between source and destination nodes under various link failure scenarios.* In this section, we first provide a graph representation of the network and then describe how we use and expand NetDice's design to explore various network states and compute the reachability probability between two nodes in each state.

1) NETWORK GRAPH REPRESENTATION

We encode the network as an edge-labeled directed graph $G = (V, E)$, such as the one presented in Figure 2, where V represents the nodes in the network, *e.g.,* routers, and E represents the directed connections between the nodes. In this representation, a physical link (e_i) is represented as a pair of *symmetrical edges* that share the same end nodes but with opposite directions, *e.g.,* Ⓢ $\overset{e_i}{\leftrightarrow}$ Ⓐ. To model random link failures, we label each edge in E with a failure rate r:

$$r : E \rightarrow \{x \in \mathbb{R} \mid 0 \leq x \leq 1\} \tag{1}$$

2) NETWORK STATE EXPLORATION

Given that every edge can be either up (working) or down (failed), we must consider $2^{|E|}$ cases to visit all failure scenarios. Accordingly, a brute-force scheme to assess the reachability property between two network nodes does not scale [23]. To avoid this state exploration overhead, NetDice [23] introduces *hot edges* whose failure alters the paths selected between two nodes by the routing and the load balancing schemes. NetDice then merges various failure scenarios that have the same set of hot edges into one network state. We refer to these states as *Equivalence Classes (ECs)* for the rest of the paper. Since all the cases grouped in one EC have the same set of potential paths between source

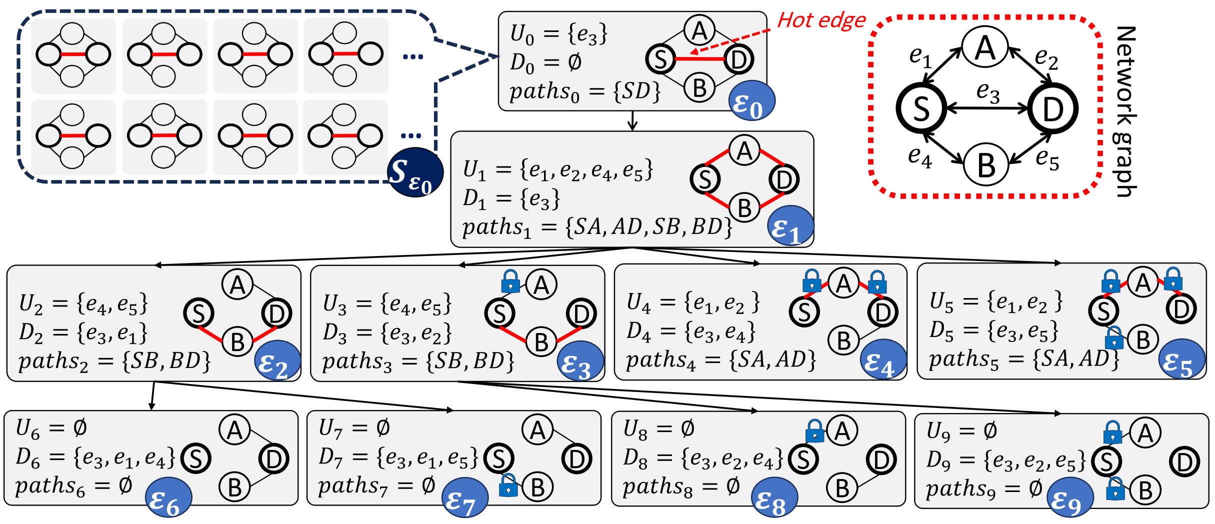

FIGURE 2. Tempus creates an exploration tree form the network graph in the functional verification phase, every node being an EC, and augments it with path information to perform temporal verification.

and destination, they are considered the same regarding the reachability property. Therefore, NetDice iterates over ECs instead of single failure scenarios to shrink the exploration space.

To explore all the equivalence classes, NetDice creates an *exploration tree* by starting from the EC with no link failure and recursively failing certain hot edges to explore other ECs. NetDice's main goal is to verify qualitative properties such as reachability. Therefore, it defines ECs based on only the set of hot edges. In contrast, Tempus aims to verify latency, a quantitative property. Thus, to be able to reason about latency, Tempus expands NetDice's state exploration algorithm by augmenting every EC with the set of potential paths between source and destination given the failed links set.[2] Specifically, we define an Equivalence Class ($\mathcal{E}_i$) as a 3-tuple (U_i, D_i, $paths_i$) in which U_i, D_i, and $paths_i$ refer to the set of hot edges that are up, the set of hot edges that are down, and the potential paths between the source and destination node, respectively.[3] Here, i is the index we use to refer to distinct ECs and their corresponding link status and path information. Using this new notion of EC, we then form the exploration tree (τ). We put the EC that corresponds to the network with no hot edge failure, *i.e.,* $D_0 = \emptyset$, as the root of the tree. To generate the rest of the tree, we recursively create children nodes by failing one of the hot edges of their corresponding parent node. We make sure not to generate the same equivalence class multiple times. Specifically, similar to NetDice, when exploring a node's children, we *lock* the hot edges that have been considered failed in previously explored children (Figure 2). While exploring a node, Tempus avoids generating an EC twice by refraining from failing locked edges.

[2]In its temporal verification phase, Tempus exploits the path information added to each EC to approximate the latency experienced by packets as they traverse various paths from the source to the destination under distinct ECs.

[3]The control plane generates $paths_i$ based on U_i and D_i.

3) FUNCTIONAL PROBABILITY CALCULATION

The exploration tree provides the set of all equivalence classes, represented as tree nodes, that should be considered while assessing both the functional and temporal properties. To compute the functional probability, *i.e.,* the probability of the destination being reachable from the source under distinct failure scenarios, we calculate the probability of each tree node ($\mathcal{E}_i$) by multiplying the probability of all the edges in U_i being up, the probability of all the edges in D_i being down, and the probability of the locked edges, that are not already included in U_i, being up. Unlike NetDice which computes the final reachability probability by summing up the probability of all equivalence classes, we use the functional probability of individual ECs ($P_F(\mathcal{E}_i)$) to compute the bounded reachability probability.

Illustrative Example: Figure 2 illustrates the exploration tree generated by Tempus while performing functional verification between Ⓢ and Ⓓ in an example network graph. In this example, we use Open Shortest Path First (OSPF) and Equal Cost Multiple Path (ECMP), two techniques that are widely deployed in modern networks, as the IP routing protocol and load balancing mechanism, respectively. We assume that the failure rate for every link (r) is 10%[4] and the weight for all the links is set to 1 while deploying OSPF. We want to compute the reachability probability from Ⓢ to Ⓓ under various failure scenarios. At the root of the tree, *i.e.,* the equivalence class with no hot edge failure ($\mathcal{E}_0$), OSPF produces *SD* as the shortest path between Ⓢ and Ⓓ. As explained previously, every EC represents all failure scenarios in which the paths selected between source and destination by the routing and load balancing schemes, *i.e.,* the set of hot edges, are the same. For instance, in $\mathcal{E}_0$, only *SD* is considered a hot edge

[4]The 10% failure rate in this illustrative example is used for ease of exposition. For our evaluations in §IV, we use the failure rates reported from production networks in [23].

since the packets are forwarded through *SD* as long as it is up. Thus, $\mathcal{E}_0$ represents all the failure scenarios shown in $S_{\mathcal{E}_0}$. The probability of $\textcircled{D}$ being reachable from $\textcircled{S}$ in this equivalence class ($P_F(\mathcal{E}_0)$) is calculated as the probability of all failure scenarios in which *SD* is up. Since the failure rate of each link is 10%, this probability is $1 - 0.1 = 0.9$.

For $\mathcal{E}_1$, we assume that the load balancing and routing techniques may send the packets via (e_1, e_2) and (e_4, e_5). Thus, $P_F(\mathcal{E}_1)$ is the probability of reaching $\mathcal{E}_1$ and $\textcircled{S}$ and $\textcircled{D}$ being reachable, *i.e.,* the probability of all failure scenarios in which *SD* is down and all other links are up.

$$P_F(\mathcal{E}_1) = 0.1 \times (0.9)^4 = 0.06561 \tag{2}$$

Similarly, the reachability probability of $\mathcal{E}_2$ is 0.0081. Since *SA* is considered locked in $\mathcal{E}_3$, $P_F(\mathcal{E}_3)$ is the probability of reaching state $\mathcal{E}_3$ and $\textcircled{D}$ being reachable from $\textcircled{S}$, *i.e.,* the probability of *SA*, *SB*, and *BD* being up and *SD* and *AD* being down.

$$P_F(\mathcal{E}_3) = (0.1)^2 \times (0.9)^3 = 0.00729 \tag{3}$$

Using the same computation, $P_F(\mathcal{E}_4)$ and $P_F(\mathcal{E}_5)$ are equal to 0.0081 and 0.00729, respectively. For cases, such as $\mathcal{E}_4$ and $\mathcal{E}_5$, in which there is an overlap between U_i and the set of locked edges, we make sure not to double count a link while computing the functional probability. For instance, while calculating the probability of locked edges being up in $\mathcal{E}_5$, we only consider *SB* since *SA* and *AD* are already covered in U_5. In $\mathcal{E}_6$, $\mathcal{E}_7$, $\mathcal{E}_8$, $\mathcal{E}_9$ the functional property does not hold since $\textcircled{D}$ is not reachable from $\textcircled{S}$.

Next, Tempus uses the reachability probability of each equivalence class to perform its temporal verification phase and obtain the bounded reachability probability (P_T).

B. TEMPORAL VERIFICATION

In the temporal verification phase, Tempus answers this question: What is the probability of bounded reachability (P_T), *i.e.,* the destination being reachable from the source within the given latency bound under various link failure scenarios? To this end, Tempus uses empirical delay measurements, *e.g.,* [8], [48], and [49], to map latency distributions to all the links in all potential paths between a source and destination pair. Then, it applies convolution on the acquired link latency distributions to obtain every path's latency distribution. After achieving a path's latency distribution, Tempus uses the distribution's CDF to compute the bounded reachability probability of the path ($P_T(\rho)$), *i.e.,* the probability of a packet traversing the path below the latency bound. Lastly, Tempus calculates each EC's bounded reachability probability ($P_T(\mathcal{E}_i)$) by running a weighted average over its potential paths' probabilities, based on the routing and load balancing algorithms, and multiplying the result by the EC's functional probability ($P_F(\mathcal{E}_i)$). The final temporal probability (P_T) is calculated as the sum of the bounded reachability probabilities of all ECs.

1) MODELING LATENCY FOR INDIVIDUAL COMPONENTS

To measure the latency distribution of a component at the core of the network, we label each edge in the network graph with various sources of delay such as *link transference delay* (l_t), *i.e.,* the time taken for a packet to be transmitted to the link and propagated through it, and *queueing delay* (l_q), *i.e.,* the time a packet spends in a node's queue before being transmitted.

To obtain the transference and queueing delay distributions, we map univariate distributions, each involving only one random variable, to empirically measured delays. Specifically, to encode transference delay, we define a function $l_t : E \rightarrow \mathcal{D}$ where $\mathcal{D}$ is a continuous univariate distribution with non-negative values. Since each physical link has the same transference delay in both directions, two symmetrical edges share the same transference delay distribution. We model queueing delay as $l_q : E \rightarrow \mathcal{D}$. Unlike transference delay, two symmetrical edges might have two different queueing delay distributions since they represent two different queues.

While designing Tempus, we assume that l_t and l_q are independent and identically distributed (*i.i.d.*) for every edge. Accordingly, after measuring l_t and l_q of an edge, Tempus uses convolution, *i.e.,* a technique for calculating a random variable whose distribution is obtained from the summation of multiple independent random variables, to obtain the latency distribution of the edge.[5] Specifically, for every hot edge (h_i) in an equivalence class, Tempus computes its latency distribution $l(h_i)$ as follows:

$$l(h_i) = l_t(h_i) * l_q(h_i) \tag{4}$$

Tempus relies on real-world empirical data to obtain latency distributions for each component and uses convolution to derive the latency distributions. However, *a closed-form solution of a convolution is usually only available for two distributions of the same type*. Specifically, while it is generally possible to convolve two exponential distributions using analytical expressions, doing the same is impractical when dealing with empirical distributions derived from real-world data due to the lack of a well-defined mathematical expression. Therefore, we cannot use analytical expressions to convolve the empirical delay measurements' distributions.

To address this, we use DIRECT [50], a method that numerically approximates the mixture of two distributions that cannot be convolved otherwise. We chose DIRECT due to its bounded error property, *i.e.,* guaranteeing that the computed distribution and the correct theoretical distribution have an error rate below a certain bound.

2) OBTAINING THE TEMPORAL PROBABILITY

We also use convolution on the latency information of all edges in a path to obtain the latency distribution of the path. For instance, given a path (ρ) containing n edges ($\{h_1, h_2, \ldots, h_n\}$), we calculate its latency distribution ($\mathcal{L}_\rho$)

[5]Convolution operation is denoted with the asterisk symbol ($*$).

as follows:

$$\mathcal{L}_{\rho} = l(h_1) * l(h_2) * \ldots * l(h_n) \tag{5}$$

Similar to the previous step, Tempus relies on DIRECT for applying convolution on the edge latency distributions. After deriving the latency distribution of a path, its time-bounded reachability probability ($P_T(\rho)$) is calculated as $cdf(\mathcal{L}_p, t)$, *i.e.,* the CDF of the path's latency distribution at time t, t being the latency bound. To compute the probability of bounded reachability for an equivalence class, we average over the bounded reachability probability of all the paths in that EC and multiply the result by the EC's functional reachability probability ($P_F(\mathcal{E}_i)$). To compute the average of path probabilities, we assign a weight to every path in the EC based on the routing and load-balancing paradigms and apply a weighted average over their probabilities. Finally, to compute the overall bounded reachability probability between two nodes, Tempus sums up the calculated probabilities obtained from performing the algorithm above over all ECs.

Example: Assume that we want to compute the bounded reachability probability between Ⓢ and Ⓓ in Figure 2 and $P_T(\rho)$ of all the paths, acquired from their corresponding latency distributions' CDF, is 0.3. We first have to calculate $P_T(\mathcal{E}_i)$ for all equivalence classes. Assuming that the weights of all the paths are equal, the bounded reachability probability of $\mathcal{E}_1$ is calculated as in Equation 6.

$$\begin{aligned} P_T(\mathcal{E}_1) &= P_F(\mathcal{E}_1) \times [0.5 \times P_T(SAD) + 0.5 \times P_T(SBD)] \\ &= P_F(\mathcal{E}_1) \times [0.5 \times 0.3 + 0.5 \times 0.3] \\ &= 0.06561 \times 0.3 = 0.019683 \end{aligned} \tag{6}$$

Similarly, $P_T(\mathcal{E}_0)$, $P_T(\mathcal{E}_2)$, $P_T(\mathcal{E}_3)$, $P_T(\mathcal{E}_4)$, and $P_T(\mathcal{E}_5)$ are 0.27, 0.00243, 0.002187, 0.00243, and 0.002187. Accordingly, P_T is computed as in Equation 7:

$$\begin{aligned} P_T &= 0.27 + 0.019683 + 2 \times (0.00243) + 2 \times (0.002187) \\ &= 0.298917 \end{aligned} \tag{7}$$

To summarize, in this section, we presented the design of Tempus, a framework that probabilistically verifies the time-bounded reachability between two nodes. While performing verification, Tempus keeps the exploration overhead low by only iterating over the equivalence classes. In the next section, we propose two optimization techniques to further reduce the verification time.

III. REDUCING VERIFICATION TIME WITH CACHING

We make two observations while designing the temporal verification phase of Tempus:

1) Multiple ECs in the exploration tree might have the same set of potential paths, *i.e.,* there might exists some $i, j \in \mathbb{Z}_{\geq 0}$ in a way that $i \neq j$ and $paths_i = paths_j$ (i and j being indexes of the equivalence classes to which $paths_i$ and $paths_j$ correspond, respectively).
2) The ECs' sets of potential paths are not necessarily mutually exclusive, *i.e.,* there might exists some $i, j \in \mathbb{Z}_{\geq 0}$ such that $i \neq j$ and $paths_i \cap paths_j \neq \emptyset$.

Based on these observations, we deploy two caching mechanisms to reduce Tempus's verification times. In particular, we cache the latency of each path the first time it is observed while iterating through the exploration tree and use the cached data if the path is observed again. Similarly, we also cache the latency of the *paths* set corresponding to every EC and use the cached data if the same set is observed in another EC. These caching mechanisms create extra memory overhead. However, in §IV, we show that by preventing repetitive tasks, such as re-computing the latency of a path shared between various ECs, and the state-keeping required for them, these optimization mechanisms might eventually improve Tempus's overall RAM consumption. For instance, in an 8-ary fat-tree, Tempus's memory consumption improves by 21% due to the optimization techniques explained below as they reduce the number of convolutions by 87×.

A. CACHING PATH LATENCY

Depending on the routing protocol and load balancing scheme, a set of potential paths ($paths_i$) is assigned to each equivalence class ($\mathcal{E}_i$). Considering that the path sets for various ECs are not mutually exclusive, *e.g., $paths_1$* and *$paths_2$* in Figure 2 share *SB* and *BD*, we can reduce the exploration time by caching each path's latency and using the cached value the next time the path is observed as we iterate over other ECs. For this purpose, we store every path between source and destination and its corresponding latency in a hash map data structure due to its fast search mechanism.

Tempus applies a weighted average on the latency of potential paths in an EC to compute its latency. The weight assigned to a path depend on the routing and load-balancing mechanisms and might be different from one EC to the other. Therefore, we store a path's latency irrespective of its weight and apply weights on the stored value based on the EC for which we are computing the latency. In §IV, we observe that caching the paths' latencies improves the exploration time by 315× in an 8-ary fat-tree.

B. CACHING EQUIVALENCE CLASS LATENCY

In the exploration tree, multiple ECs may have the same *paths* set, *e.g.,* $\mathcal{E}_2$ and $\mathcal{E}_3$ in Figure 2. Accordingly, to further reduce the exploration time, we cache the *paths* set as we iterate over an EC. This way, while iterating over an equivalence class whose *paths* has already been cached, Tempus uses the cached data instead of re-computing the EC's latency. Similar to caching path latency, we use a hash map data structure for this purpose. We show in §IV that, in an 8-ary fat-tree, caching the latency of the potential paths sets corresponding to different equivalence classes reduces the exploration time of Tempus by 4×.

IV. EVALUATION

We implement Tempus in 600 lines of Julia code [51].[6] Given the topology, a source and destination node, the queueing and transference delay measurements, and a time

[6]Tempus's codebase is available at https://github.com/hopnets/Tempus

threshold (latency bound), our prototype outputs the temporal probability, *i.e.,* the time-bounded reachability probability, under various failure scenarios. In this section, we use the term "Tempus" to refer to our design with the proposed caching mechanisms, and explicitly mention whenever we are referring to Tempus without any of the optimization schemes. We run our prototype on a machine with 70 GB of RAM and a 6-core CPU (12 logical processors). Unless stated otherwise, we report the results for cases in which Tempus is running on one logical processor. In summary, our findings show that Tempus is accurate and scalable. Specifically, Tempus precisely captures the upper bound on the 99th percentile end-to-end latency under various degrees of load. Furthermore, Tempus verifies the bounded reachability between all $(source, destination)$ pairs in an 8-ary fat-tree with 100 Gbps links in 8 minutes and 32 seconds. In comparison, simulating all failure scenarios (all possible ECs) in an 8-ary fat-tree on the same machine using Parsimon [52], a state-of-the-art parallel network simulator, takes three orders of magnitude longer.

A. EXPERIMENT SETUP

1) TOPOLOGY

Table 1 presents the topologies used in our experiments. We test Tempus on various scales of Wide Area Networks (WAN) and datacenter networks. In particular, for WAN topologies, we select Latnet, Highwinds, AT&T, Uninett, and GtsCe from Topology Zoo [53]. For datacenter networks, we implement a 2 × 3 two-tier leaf-spine topology, *i.e.,* two spine routers connected to three leaf routers, and various scales of the fat-tree [54] topology. A k-ary fat-tree datacenter topology is a three-tiered network with $\frac{k^2}{4}$ core routers connected to k pods each containing $\frac{k}{2}$ aggregate routers and $\frac{k}{2}$ edge routers. The edge routers are connected to servers and the aggregate routers are connected to both core and edge routers.

TABLE 1. List of topologies on which Tempus is tested.

Type	Topology	# of routers	# of links
Datacenter networks	2×3 leaf-spine	5	6
	Fat-tree ($k = 4$)	20	32
	Fat-tree ($k = 8$)	80	256
	Fat-tree ($k = 16$)	320	2048
	Fat-tree ($k = 32$)	1280	16384
WAN	Latnet	69	74
	Highwinds	18	31
	AT&T	25	56
	Uninett	74	101
	GtsCe	149	193

2) EXPLOITING EMPIRICAL QUEUEING AND TRANSFERENCE DELAY MEASUREMENTS

In our experiments, unless stated otherwise, we assume that the link bandwidths are 100 Gbps [17], [55], [56], [57], [58], [59]. Accordingly, to model the queueing delay, we scale the TCP and DCTCP queueing delay measurements reported in [8] based on the link rate. To model the transference delay of the datacenter and wide area networks, we use the propagation delay measurements in [48] and [49], respectively, and increase them by the transmission delay of one MTU (1500 bytes).[7] Table 2 provides details about the mapped queueing and transference delay distributions. Note that Tempus's design makes no assumption about the shapes and characteristics of these distributions. Accordingly, the queueing and transference distributions can be tuned by the operators based on their traffic characteristics. To show this, in §IV-B, we also test Tempus with uniformly distributed transference delays and queueing delays obtained from network simulations.

TABLE 2. For our evaluations, we map univariate distributions to the data provided in [8], [48], and [49] to generate queueing, WAN transference, and DC transference delay distributions, respectively.

Category ↓	Mean	50th percentile	99th percentile
DC transference (ns) [48]	385	321	853
WAN transference (ns) [49]	2870575	2775120	6319120
TCP queueing (ns) [8]	35380	35137	45352
DCTCP queueing (ns) [8]	2152	2244	2413

3) EVALUATED METRICS

We report the functional verification time, the temporal verification time, and the temporal probability (*i.e.,* time-bounded reachability probability) as the main evaluation metrics for Tempus. We also record more fine-grained metrics such as the number of convolutions required in the temporal verification phase.

4) PARAMETER SETTINGS

In our experiments, we deploy Open Shortest Path First (OSPF) as the routing protocol and Equal Cost Multiple Path (ECMP) as the load-balancing paradigm. Since the main focus of Tempus's design is temporal verification, for simplicity of this section's experiments and similar to previous work [23], we set the failure rate of every edge in the network graph to 0.1% and set all link weights to 1 for OSPF routing. We set the source and destination nodes to two randomly selected edge routers and use two cutoff thresholds while running the experiments, *i.e., the accuracy level* $(1 - 10^{-\delta})$ and *the timeout threshold* (γ). We terminate an experiment if the total run time exceeds γ. Additionally, while performing functional verification, if the reachability probability of a node falls below $10^{-\delta}$, we refrain from further exploring its children in the exploration tree. Unless stated otherwise, we set γ and δ to two hours and 6, respectively.

B. EXPERIMENT RESULTS

We compare Tempus's verification time with state-of-the-art network simulators, validate its accuracy, and evaluate it under various network scales and latency distributions.

[7] In a 100 Gbps network, transmitting 1500 bytes to the link takes 120 nanoseconds.

TABLE 3. State-of-the-art network simulators are not suitable for verifying latency under all failure scenarios. "αy βmo γd δh ηmin κs" represents α years, β months, γ days, δ hours, η minutes, and κ seconds.

Tool → Fat-tree scale (k) ↓	Tempus	OMNeT++ [13]	Parsimon [52]	DONS [15]
4	$2min$	$29d$ $20h$ $55min$ $10s$ ($21,508\times$ improvement)	$4h$ $15min$ $11s$ ($128\times$ improvement)	$14d$ $11h$ $38min$ $57s$ ($10,430\times$ improvement)
8	$8min$ $32s$	$15y$ $2mo$ $18d$ $12h$ $13min$ $20s$ ($924,498\times$ improvement)	$1mo$ $5d$ $5h$ $9min$ $40s$ ($5,942\times$ improvement)	$7y$ $8d$ $5h$ $46min$ ($426,641\times$ improvement)
16	$1h$ $53min$ $36s$	$1820y$ $7mo$ $5d$ $2h$ $47min$ $23s$ ($8,507,792\times$ improvement)	$22y$ $1mo$ $16h$ $17min$ $31s$ ($103,206\times$ improvement)	$200y$ $1mo$ $13d$ $10h$ $48min$ $32s$ ($935,179\times$ improvement)
32	$18d$ $22h$ $49min$ $35s$	$176641y$ $3mo$ $26d$ $14h$ $20min$ ($3,355,522\times$ improvement)	$351y$ $1d$ $2h$ $15min$ $37s$ ($6,668\times$ improvement)	$5233y$ $6mo$ $8d$ $5h$ $21min$ $28s$ ($99,417\times$ improvement)

1) TEMPUS'S VERIFICATION TIME IS SIGNIFICANTLY LOWER THAN THE STATE-OF-THE-ART NETWORK SIMULATORS

To evaluate Tempus, as the first experiment, we run it on various scales of k-ary fat-tree topology with 100 Gbps links under 25% load and compare the verification time with the time taken for simulating all failure scenarios using the state-of-the-art network simulators [13], [15], [52]. For this purpose, we use OMNeT++ [13], a non-parallel packet-level network simulator, Parsimon [52], a parallel flow-level simulator, and DONS [15], a parallel packet-level simulator. Tempus and OMNeT use one logical processor and Parsimon and DONS exploit all 12 processors. Even with fast network simulators, it takes a long time to simulate all failure scenarios in large networks. Accordingly, in these experiments, we estimate the total simulation time for OMNeT, Parsimon, and DONS by simulating one failure scenario and multiplying the simulation time by the number of ECs in each topology. To provide a fair comparison, considering that these simulators simultaneously measure latency between every $(source, destination)$ pair in the network graph, we scale Tempus's verification time to approximate the time it requires to verify latency between all $(source, destination)$ pair of nodes. To this end, for each topology, we run Tempus for one $(source, destination)$ pair and multiply the verification time by $\binom{N}{2}$ (N is the number of edge routers to which the servers are connected).[8]

The results, presented in Table 3, illustrate that even with the state-of-the-art fast and parallel simulators, verifying latency at scale and under all failure scenarios is impractically slow. Tempus enables this critical network management task by accelerating it, e.g., in an 8-ary fat-tree topology, Tempus only requires 8 minutes and 32 seconds, compared with approximately 15 years, one month, and 7 years (!) required for OMNeT, Parsimon, and DONS, respectively.[9] That is, Tempus's verification time is $924,498\times$, $5,942\times$, and $426,641\times$ lower than OMNeT, Parsimon, and DONS, respectively. Tempus's speedup over these simulators is due to the fact that simulating all the entities in a network and every packet's/flow's life cycle consumes considerably more time than computing the distribution of the extra latency imposed by various network components and aggregating them.

[8]A k-ary fat-tree network has $\frac{k^2}{2}$ edge routers.

[9]Note that, despite being faster than DONS, Parsimon only estimates flow tail latency and does not cover packet-level latency metrics such as average RTTs.

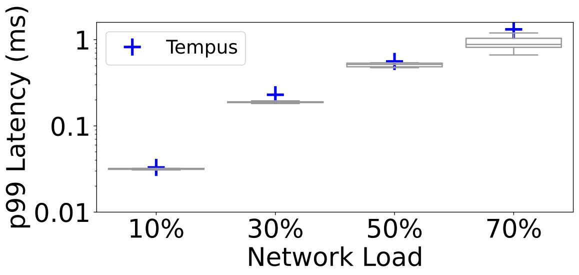

FIGURE 3. Tempus accurately approximates the 99th percentile delay. The box and whisker plots represent distinct tail latencies recorded as we repeat the simulations multiple times.

2) TEMPUS ACCURATELY APPROXIMATES THE NETWORK DELAY

As part of its temporal verification phase, Tempus approximates the delay imposed by network components in all failure scenarios, *i.e.,* ECs, and uses the approximations to estimate the overall network delay packets experience as they traverse from the source to the destination. To evaluate the accuracy of Tempus's approximations, we simulate the no-failure scenario in a two-tier leaf-spine topology using OMNeT++. Specifically, we simulate two spine routers, three leaf routers, and ten machines connected to each leaf and select one server as the source and one as the destination (the source and the destination are connected to two distinct leafs). For one second, while other machines inject various degrees of load into the network, the soure sends one packet toward the destination every millisecond. At the receiver, we capture the packets sent by the source, record the delay they have experienced while traversing the network, and measure the tail (99^{th} percentile) packet-level delay. We repeat the simulations ten times to capture various randomnesses caused by the simulator. For Tempus, we input the network topology and the empirical queueing time measurements from our simulations and record the tail temporal probability estimated by Tempus for the no-failure scenario. Figure 3 compares Tempus's output with the tail latencies observed from the simulations under various degrees of load. We observe that, irrespective of the load, Tempus accurately captures the upper limit for the tail delay. Specifically, with various degrees of load, the tail delay observed under distinct repetitions does not surpass Tempus's approximation of the 99^{th} percentile packet-level delay, making Tempus an effective tool for SLA verification.

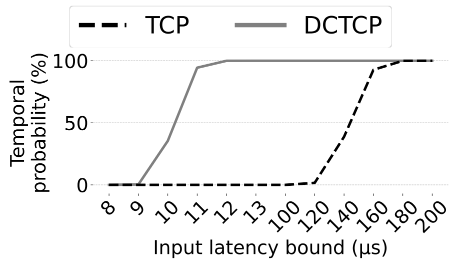

FIGURE 4. **Using Tempus, we verify that DCTCP's latency is much lower than TCP.**

3) TEMPUS'S OUTPUT ILLUSTRATES DCTCP's PERFORMANCE SUPERIORITY OVER TCP

To further evaluate Tempus's correctness, we investigate if its output is consistent with the latency trends of existing proposals. In [8], Alizadeh et al. design a congestion control protocol, *i.e.,* DCTCP, that exploits Explicit Congestion Notification (ECN) to provide a faster reaction to congestion compared to TCP. They illustrate that DCTCP significantly improves the latency over TCP. In this experiment, we run Tempus on a 16-ary fat-tree topology with both TCP's and DCTCP's empirical queueing delay measurements, reported in [8], and measure the temporal probability under various input latency bounds. Figure 4 illustrates that Tempus highlights DCTCP's superiority over TCP. In particular, with DCTCP, we achieve the temporal probability of ~ 1.0 for a latency bound that is $16.7\times$ lower compared to the case with TCP showing that DCTCP has significantly lower latency than TCP under all failure scenarios.

4) UNDER SMALL AND LARGE NETWORKS, TEMPUS DEDICATES MOST OF THE TIME TO TEMPORAL AND FUNCTIONAL VERIFICATION, RESPECTIVELY

We evaluate Tempus's functional and temporal verification time under various scales of WAN and datacenter networks presented in Table 1. For the rest of the paper, unless stated otherwise, we report the verification times for running Tempus once, *i.e.,* the times are not scaled to cover verifying latency for all $(source, destination)$ pairs. The results, illustrated in Figure 5, show that when the scale of the network is small, Tempus dedicates most of the verification time to temporal verification. For instance, in an 8-ary fat-tree, the time spent on temporal verification is $21\times$ larger than the time dedicated to functional verification. Under large networks, on the other hand, the majority of time is spent on functional verification (*e.g.,* Tempus spends 94% of its verification time performing functional verification for a 32-ary fat-tree). The temporal verification time is mainly influenced by the number of convolutions required for obtaining the components' latency distributions. Due to Tempus's optimization mechanisms (§III), the temporal verification time is less affected by the network scale compared to the functional verification time. Particularly, as we move from 4-ary to 32-ary fat-tree in Figures 5c and 5d, the functional verification time increases by $9,075\times$ while the number of required convolutions and the temporal verification time only increase by $12\times$ and $15\times$, respectively.

5) TEMPUS'S VERIFICATION TIME INCREASES AS THE DEGREE OF PATH DIVERSITY RISES

In Figure 5a, we observe that Latnet has a lower verification time than Highwinds and AT&T networks despite having $3.8\times$ and $2.8\times$ more routers and $2.5\times$ and 32% more links, respectively. This is because Latnet provides much less path diversity between edge routers compared to Highwinds and AT&T and, thus, Tempus iterates through fewer paths and applies fewer convolutions while verifying Latnet. Particularly, compared to Highwinds and AT&T, Tempus iterates through $6\times$ and $11\times$ fewer paths and, as shown in Figure 5b, applies $6.8\times$ and $8.7\times$ fewer convolutions, respectively, when verifying the latency in Latnet.

6) TEMPUS ALSO SUPPORTS NETWORK LATENCY VERIFICATION WITH THEORETICAL DISTRIBUTIONS

In this paper, we mainly exploit empirical results from other papers [8], [48], [49]. However, our method can be deployed with theoretical distribution mapping as well. To show this, we repeat our WAN and fat-tree experiments and set the link propagation delays to $\mathcal{U}(10\mu s, 100\mu s)$ [60] and $\mathcal{U}(1\mu s, 5\mu s)$ [61], respectively.[10] In Figure 6, we observe that using different latency distributions affects Tempus's temporal verification time as it impacts the time required for applying numerical convolution on them. For instance, under 32-ary fat-tree, using uniformly distributed propagation delays causes $9\times$ larger temporal verification time compared to using the empirical propagation delays in [48]. However, despite this increase in the temporal verification time, running Tempus only takes ~ 77 minutes.

7) TEMPUS'S VERIFICATION TIME IS NOT AFFECTED BY THE INPUT LATENCY BOUND

So far, we have illustrated that Tempus experiences distinct verification times with various network graphs and delay measurements (Figures 5 and 6, respectively). In particular, increasing the degree of path diversity between source and destination results in larger verification times. Additionally, since the time required for applying numerical convolution on two distributions differs as we change the distributions, using various transference and queueing delay measurements alters the temporal verification time.

We also quantify the effects of the third input, *i.e.,* latency bound, on Tempus's verification time. Specifically, we run Tempus with various latency thresholds on 8-ary, 16-ary, and 32-ary fat-tree topologies. Figure 7 presents the results. We observe that, unlike the other two inputs, *i.e.,* network graph and empirical delay measurements, changing the latency bound does not impact Tempus's verification time.

[10]$\mathcal{U}(\alpha, \beta)$ denotes a uniform distribution between α and β.

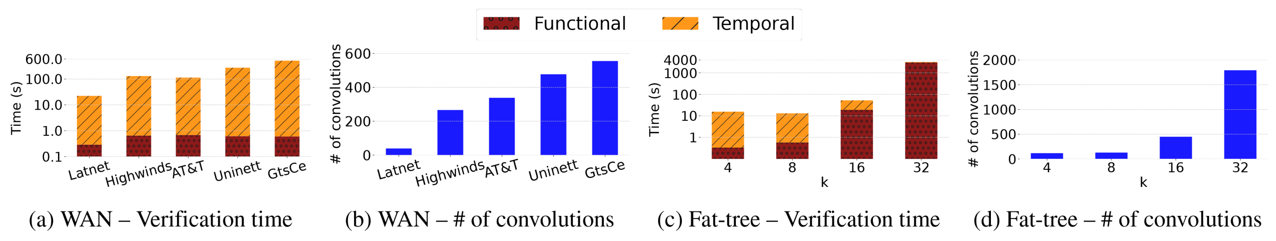

FIGURE 5. **Tempus verifies all the topologies from Table 1 in a short time.**

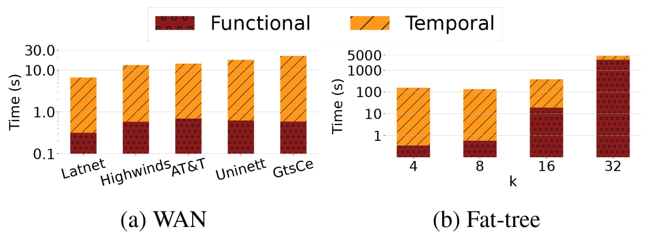

FIGURE 6. **Tempus can work with theoretical distributions.**

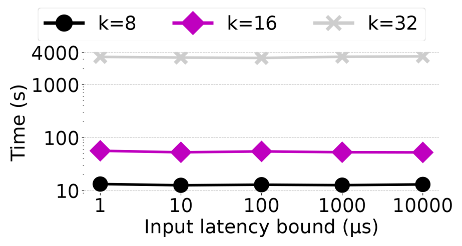

FIGURE 7. **Changing the latency bound does not affect Tempus's verification time.**

This is because the latency threshold is only used to calculate a path's latency, from the CDF of the distribution mapped to it, in the temporal verification phase. Since the value of the latency threshold does not affect the run-time of this step, altering the latency threshold has no effect on the verification time.

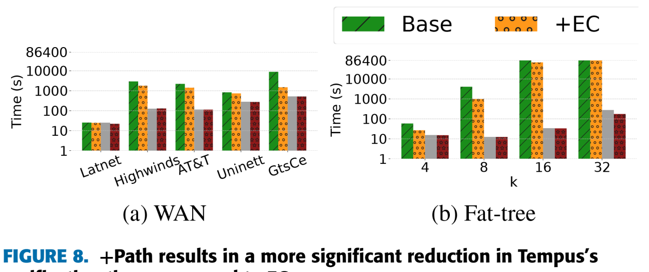

FIGURE 8. **+Path results in a more significant reduction in Tempus's verification time compared to EC.**

C. TEMPUS DESIGN DEEP-DIVE

1) THE IMPACT OF THE OPTIMIZATION SCHEMES ON TEMPUS'S VERIFICATION TIME

In §III, we described two caching mechanisms deployed in Tempus to reduce its verification time. To measure the extent to which each mechanism reduces the verification time, we run the WAN and fat-tree experiments under four scenarios: 1) enabling no caching scheme (Base), 2) only storing the latency of ECs (+EC), 3) solely caching path latency (+Path), and 4) enabling both caching schemes (+Path+EC). These optimizations exclusively focus on minimizing the temporal verification time. Accordingly, in Figure 8, we only report the time Tempus dedicates to temporal verification under various scenarios. For these experiments, we set the timeout threshold (γ) to 24 hours. Our findings show that, while caching the latency corresponding to both ECs and paths improves the temporal verification time, storing paths' latency has a more significant impact than storing ECs' latency. For instance, with a 32-ary fat-tree, both *Base* and *+EC* scenarios hit the 24-hour time threshold. *+Path*, on the other hand, results in $\sim$ 5-minute and $\sim$ 56-minute temporal and total verification times, respectively.

2) THE MEMORY CONSUMPTION OVERHEAD

Storing the path and EC latency information creates memory footprint. At the same time, caching these information prevents Tempus from repetitively executing operations, such as applying convolution, which in turn reduces the memory overhead. To investigate this trade-off, we report Tempus's average and maximum RAM consumption, under various optimization paradigms and distinct network scales. Particularly, while running an experiment, we run a process on another thread that records the *VmRSS*, *i.e.,* Virtual memory Resident Set Size, of the experiment every second. We run this experiment for 4-, 8-, 16-, and 32-ary fat-tree networks. Figure 9 illustrates Tempus's mean and maximum VmRSS under distinct optimization levels. For smaller scales of networks, caching the path and EC latency information reduces the memory consumption in addition to decreasing the verification time. For instance, in an 8-ary fat-tree, *+Path+EC* reduces the mean RAM consumption by 21% compared to *Base* due to reducing the number of required convolutions by 87×. With large networks, on the other hand, using caching increases the overall memory consumption. Specifically, under a 32-ary fat-tree, *+Path+EC* increases the average VmRSS by 79% compared to *Base*. However, the increase experienced for a 32-ary fat-tree is still quite low considering the RAM capacity of modern machines and also compared to the memory overhead of network simulators. For example, we observe $\sim$ 15 GB maximum RAM consumption while simulating a 32-ary fat-tree network in DONS which is $\sim$ 2.5× higher than *+Path+EC*.

3) CUTOFF THRESHOLD ANALYSIS

We set two parameters to ensure low verification time and resource consumption: 1) the timeout threshold (γ) and 2) the

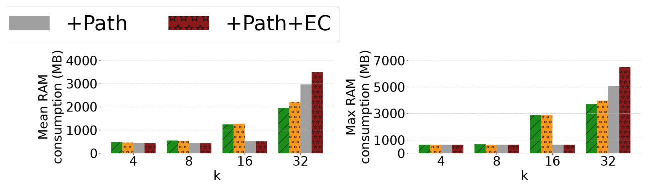

FIGURE 9. Tempus imposes low memory overhead.

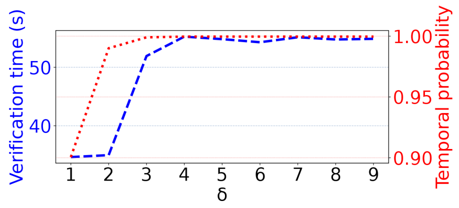

FIGURE 10. The temporal probability converges when δ passes 3.

accuracy level $(1-10^{-\delta})$. So far, we observed that, with $\delta = 6$ and a γ larger than 2 hours, the timeout threshold is never hit as long as the optimization schemes are enabled. There is a trade-off between choosing small and large δ values. Small δ values result in inaccuracy in Tempus's output while, with large δ values, Tempus would iterate over numerous ECs with minuscule functional and temporal probabilities and experience high verification times. To analyze the impact of δ on Tempus's correctness and verification time, we run our 16-ary fattree experiments while setting γ to 2 hours. We use a large latency threshold ($T = 200\mu s$) in which the temporal probability is expected to be ~ 1, *i.e.,* packets are transmitted below T as long as the source and destination are reachable. Figure 10 illustrates that Tempus's output, *i.e.,* bounded reachability probability, converges to the correct result as δ passes 3. We also observe that, as we increase δ from 3 to 4, the verification time increases by 6.4% while the temporal probability only grows by 0.06%. This is because Tempus iterates over more ECs, which have low functional and temporal probability values, as we increase the accuracy. Our experiments throughout this paper show that, even with large δ values such as 6, Tempus remains scalable and provides short verification times.

V. LIMITATIONS AND FUTURE DIRECTIONS

A. FAILING NODES IN ADDITION TO LINKS

In this paper, we assume that link failures are independent. In practice, however, there are sometimes interdependencies between failures, *e.g.,* when a router fails. It is possible to extend our method to model such failures. For example, to model router failures, in addition to hot edges, we should also define the concept of *hot nodes*, *i.e.,* the nodes located on the paths between source and destination chosen by the routing protocol and load balancer, and define a network state as the set of hot edges and hot nodes. Then, to create the exploration tree, we explore both hot edge failure and hot node failure scenarios. While failing a hot node, we add the group of links corresponding to it to the list of failed links. We leave expanding Tempus's design to support both link and node failure to future work.

B. COVERING OTHER SOURCES OF LATENCY

In this paper, we focus on modeling the queueing, the propagation, and the transmission delay as the main sources of latency at the network core. However, Tempus's design principles are also applicable to other sources of delay such as middleboxes and various layers of the host networking stack. For instance, to encompass the latency imposed by the host networking stack, it suffices to obtain the latency distribution of each layer and compute their convolution. In general, Tempus's design can support any source of latency as long as two conditions hold: 1) the availability of either the empirical delay measurements or the statistical distribution of the source of latency and 2) the independence of different sources of latency. We leave designing a latency verifier in scenarios where either of these conditions are not met for future work.

C. DYNAMICALLY VERIFYING LATENCY

Tempus targets static network verification, *i.e.,* given the network graph and delay measurements, it verifies the latency under various failure scenarios. However, the traffic characteristic and the shape of delay measurements can alter from time to time or due to events such as congestion and component failure which requires Tempus to dynamically tune its distributions. It is possible to dynamically update our model by constantly sending probe messages, *e.g.,* traceroute, into the network and tuning the delay distributions accordingly, but this would create an extra overhead on both the network and the end hosts [12]. We leave tailoring Tempus's design for dynamic latency verification to future work.

D. VERIFYING MULTIPLE QUANTITATIVE PROPERTIES

In this work, we focus our design on packet-level network latency verification. However, we believe that Tempus's design can be extended to encompass other quantitative properties such as throughput. Specifically, given the empirical link utilization measurements, one can extend Tempus to probabilistically verify the maximum achievable throughput from a source to a destination node under various failure conditions. We leave expanding Tempus's design to verify multiple quantitative properties to future work.

VI. RELATED WORKS

The proposals on network verification can be broadly categorized into two groups: 1) *Qualitative verifiers* that analyze the network's functional correctness such as the reachability between two nodes, and 2) *Quantitative verifiers* that assess properties such as bandwidth and latency.

A. QUALITATIVE VERIFICATION

Many proposals in network verification focus on qualitative verification and answer the following questions: Is node "A" reachable from node "B"? [22], [24], [27], [29], [30], [31], [32], [33], [40], [41], [42], [43], [62], [63], [64], is a path between two nodes loop-free? [22], [29], [30], [31],

[34], [35], [41], [43], [62], [63], [65], is the reachability preserved under various failure scenarios? [23], [24], [27], [31], [32], [36], [37], [63], [65], [66], is the first and last hop of every path in a datacenter an edge switch? [28], are all the affected hosts eventually identified by the intrusion detection system? [21], will the route received from a neighboring router eventually be sent to other neighbors? [26] etc. Such verifiers are orthogonal to temporal verifiers such as Tempus that reason about important quantitative properties such as latency.

B. QUANTITATIVE VERIFICATION

Networks have increasingly strict performance requirements [7], [8]. Despite being vastly used, existing network simulators [13], [14], [15], [52] are slow for verifying performance under all network states (§IV). Accordingly, some proposals [4], [5], [9], [10], [11], [12], [38], [39], [44], [46], [47], [67] have focused on tailoring network verification frameworks for quantitative reasoning. Such proposals focus on performance properties such as bandwidth, throughput, and latency, and try to answer questions like does a server get overloaded under a certain input load? [9], is the throughput requirements met given the traffic characteristics? [12], do datacenter networks suffer from link overload under various failure scenarios? [5], [68], how severe is the overload, if any? [69], and what is the probability of the network being overload-free under various failure scenarios? [4]. A subset of qualitative verifiers that are closest to Tempus focus on latency as the quantitative verification property [9], [10], [11], [12], [38], [39], [44], [45], [46], [47], [70], [71]. These proposals either do not scale well, only verify a specific aspect of the network, are coarse-grained, or do not consider link failures. We next summarize their limitations and distinctions with Tempus.

C. STATIC NETWORK LATENCY VERIFICATION

Many latency verifiers statically model the network and traffic characteristics to investigate if the latency requirements are met. Unlike Tempus, these verifiers are not general and are optimized for special purposes by partially modeling specific aspects of the network [10], [11], [46], [47]. For instance, LogNIC [47], a framework that uses data flow modeling [72] to analyze latency, models only smart NICs and does not take the latency imposed by load balancers and link failures into consideration. PIX [10], NF-SE [46], and CBS [38] focus on verifying the performance of network functions and do not model sources of latency that are not described as network functions, *e.g.,* queueing delay. Arashloo et al. [11] use formal methods to reason about network latency but only focus on the delay imposed by queues at the core of the network and do not take other sources of latency such as propagation delay into account. Additionally, unlike Tempus, their technique is *deterministic, i.e.,* it returns a yes or no answer to whether a latency requirement is met given the current topology, and does not encompass link failures. Given the probabilistic nature of networks, our position is that latency requirements can be better captured probabilistically.

Similar to Tempus, some proposals [12], [39], [44], [70] statically model the entire in-network latency. QNA [44] is among the initial works in this area and uses algebraic frameworks to perform quantitative network verification. However, QNA does not model link failures. Plus, its model does not cover statistical distributions [12]. In [39], Helm et al. use extreme value theory [73] to verify flow tail latency. Unlike Tempus, their design relies on flow-level latency measurements and, thus, is considered coarse-grained for SLAs that focus on packet-level RTTs [1], [2], [3]. SLA-verifier [12] models the latency imposed to flows by network components and use it to verify latency. To this end, SLA-verifier uses the performance counters, *e.g.,* packets/bytes counts, provided by SDN switches. Their approach has two limitations: 1) It does not model failures and 2) it verifies per-flow performance metrics and does not support packet-level latency verification. Targeting fine-grained latency verification, Larsen et al. propose WNetKAT [45], *i.e.,* a weighted version of the NetKAT algebra [74], to verify latency with packet-level granularity. Alas, WNetKAT is deterministic and does not verify latency under various failure scenarios. Tempus, on the other hand, probabilistically verifies packet-level latency while taking all link failures into account. Liu et al. [9] target the verification of distributed control in self-driving systems, their complex and non-deterministic interactions, and quantitative metrics such as latency but leave the scaling of their approach to large-scale networks to future work (*e.g.,* their technique exhausts the one-hour time budget as the scale of the k-ary fat-tree network goes beyond $k = 8$).

D. DYNAMIC NETWORK LATENCY VERIFICATION

Another group of proposals [12], [70], [71] attempt to dynamically verify network latency. Particularly, these proposals try to verify if latency requirements are met while tracking the changes in the network and traffic characteristics through time. In addition to its static verification phase, SLA-verifier also deploys a dynamic verification phase in which it uses probe messages, *e.g.,* traceroute, to update its flow-level performance model. Unfortunately, using probe messages imposes overhead on both the network and end hosts [12]. Plus, this method detects latency violations only after the network is already in an undesirable state. AalWiNes [70] dynamically verifies latency under multiple failure scenarios using weighted pushdown automata, *i.e.,* a quantitative extension of the classic automata theory. However, AalWiNes does not cover all failure scenarios, is deterministic, and is restricted to MPLS networks only. In [71], Choi et al. use Inband Network Telemetry [75] and Metric Dynamic Logic [76] to detect SLA violations. Unlike Tempus, their technique is deterministic and focuses on detecting latency violations as they happen in run-time rather than verifying latency under all failure scenarios in advance.

VII. CONCLUSION

Network verifiers and simulators strike a balance between expressiveness and speed. On one end of the spectrum, high-fidelity simulators can accurately model details of networks and validate latency but are too slow to navigate the entire state of large networks, and on the opposite end, fast verifiers comprehensively search the state space but rely on abstract models that exclude crucial aspects of networks for latency verification, especially under failure. We design Tempus, a probabilistic verification framework that analyzes latency under all failure scenarios. Given the network topology, statistical latency distributions or empirical delay measurements, and a latency bound, Tempus returns the time-bounded reachability probability. We evaluate Tempus under various network scales and show that it is fast and scalable. For instance, Tempus verifies the latency between a source and a destination node in a 16-ary fat-tree network in only 52 seconds. We also show that Tempus precisely approximates network latency.

ACKNOWLEDGMENT

(Sepehr Abdous and Senapati Diwangkara contributed equally to this work.)

REFERENCES

[1] (2023). *Verizon Global Latency and Packet Delivery SLA*. [Online]. Available: https://www.verizon.com/business/terms/global_latency_sla/

[2] (2022). *AT&T Broadband SLA*. [Online]. Available: https://www.att.com/support/smallbusiness/article/smb-internet/KM1254969/

[3] (2016). *COMCAST Service Level Agreement for Wholesale Dedicated Internet*. [Online]. Available: https://www.comcasttechnologysolutions.com/resources/service-level-agreement

[4] Y. Zhang, H. Xu, C. J. Xue, and T.-W. Kuo, "Probabilistic analysis of network availability," in *Proc. IEEE 30th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2022, pp. 1–11.

[5] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "Detecting network load violations for distributed control planes," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2020, pp. 974–988.

[6] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, "High-resolution measurement of data center microbursts," in *Proc. Internet Meas. Conf.*, Nov. 2017, pp. 1–23.

[7] G. Kumar, N. Dukkipati, K. Jang, H. M. G. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat, "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 1–23.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.*, Aug. 2010, pp. 1–26.

[9] B. Liu, A. Kheradmand, M. Caesar, and P. B. Godfrey, "Towards verified self-driving infrastructure," in *Proc. 19th ACM Workshop Hot Topics Netw.*, Nov. 2020, pp. 96–102.

[10] R. Iyer, K. Argyraki, and G. Candea, "Performance interfaces for network functions," in *Proc. NSDI*, 2022, pp. 1–26.

[11] M. T. Arashloo, R. Beckett, and R. Agarwal, "Formal methods for network performance analysis," in *Proc. NSDI*, 2023, pp. 1–26.

[12] Y. Zhang, W. Wu, S. Banerjee, J.-M. Kang, and M. A. Sanchez, "SLA-verifier: Stateful and quantitative verification for service chaining," in *Proc. IEEE Conf. Comput. Commun.*, May 2017, pp. 1–9.

[13] (2020). *Omnet++ Simulator*. [Online]. Available: https://omnetpp.org/

[14] (2020). *Ns-3 Simulator*. [Online]. Available: https://www.nsnam.org/

[15] K. Gao, L. Chen, D. Li, V. Liu, X. Wang, R. Zhang, and L. Lu, "DONS: Fast and affordable discrete event network simulation with automatic parallelization," in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 11–15.

[16] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu, "HPCC: High precision congestion control," in *Proc. ACM Special Interest Group Data Commun.*, Aug. 2019, pp. 1–26.

[17] S. Abdous, E. Sharafzadeh, and S. Ghorbani, "Practical packet deflection in datacenters," *Proc. ACM Netw.*, vol. 1, no. 3, pp. 1–25, Nov. 2023.

[18] S. Arslan, Y. Li, G. Kumar, and N. Dukkipati, "Bolt: Sub-RTT congestion control for ultra-low latency," in *Proc. NSDI*, 2023, pp. 1–18.

[19] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 1–20.

[20] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin, "Programmable packet scheduling with a single queue," in *Proc. ACM SIGCOMM Conf.*, Aug. 2021, pp. 1–12.

[21] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness verification of stateful network functions," in *Proc. NSDI*, 2020, pp. 1–17.

[22] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM Conf.*, Aug. 2011, pp. 1–20.

[23] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol.*, Jul. 2020, pp. 750–764.

[24] T. Alberdingk Thijm, R. Beckett, A. Gupta, and D. Walker, "Modular control plane verification via temporal invariants," in *Proc. PLDI*, 2023, pp. 1–18.

[25] X. Liu, P. Zhang, H. Li, and W. Sun, "Modular data plane verification for compositional networks," in *Proc. PACMNET*, 2023, pp. 1–20.

[26] A. Tang, R. Beckett, S. Benaloh, K. Jayaraman, T. Patil, T. Millstein, and G. Varghese, "Lightyear: Using modularity to scale BGP control plane verification," in *SIGCOMM*, 2023, pp. 1–13.

[27] Q. Xiang, C. Huang, R. Wen, Y. Wang, X. Fan, Z. Liu, L. Kong, D. Duan, F. Le, and W. Sun, "Beyond a centralized verifier: Scaling data plane checking via distributed, on-device verification," in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 1–26.

[28] S. Renganathan, B. Rubin, H. Kim, P. L. Ventre, C. Cascone, D. Moro, C. Chan, N. McKeown, and N. Foster, "Hydra: Effective runtime network verification," in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 182–194.

[29] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw.*, Aug. 2012, pp. 1–22.

[30] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. NSDI*, 2013, pp. 1–20.

[31] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *Proc. NSDI*, 2020, pp. 1–12.

[32] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li, "Differential network analysis," in *Proc. NSDI*, 2022, pp. 1–26.

[33] Q. Xiang, R. Wen, C. Huang, Y. Wang, and F. Le, "Network can check itself: Scaling data plane checking via distributed, on-device verification," in *Proc. 21st ACM Workshop Hot Topics Netw.*, Nov. 2022, pp. 85–92.

[34] D. Guo, S. Chen, K. Gao, Q. Xiang, Y. Zhang, and Y. R. Yang, "Flash: Fast, consistent data plane verification for large-scale network settings," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 314–335.

[35] Z. Zhou, M. He, W. Kellerer, A. Blenk, and K.-T. Foerster, "P4Update: Fast and locally verifiable consistent network updates in the P4 data plane," in *Proc. 17th Int. Conf. Emerg. Netw. EXperiments Technol.*, Dec. 2021, pp. 175–190.

[36] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 1–12.

[37] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 1–22.

[38] X. Zhang, H. Duan, C. Wang, Q. Li, and J. Wu, "Towards verifiable performance measurement over in-the-cloud middleboxes," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2019, pp. 1162–1170.

[39] M. Helm, F. Wiedner, and G. Carle, "Flow-level tail latency estimation and verification based on extreme value theory," in *Proc. 18th Int. Conf. Netw. Service Manage. (CNSM)*, Oct. 2022, pp. 359–363.

[40] T. A. Thijm, R. Beckett, A. Gupta, and D. Walker, "Kirigami, the verifiable art of network cutting," in *Proc. IEEE 30th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2022, pp. 1–12.

[41] A. Horn, A. Kheradmand, and M. R. Prasad, "A precise and expressive lattice-theoretical framework for efficient network verification," in *Proc. IEEE 27th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2019, pp. 1–12.
[42] T. Inoue, R. Chen, T. Mano, K. Mizutani, H. Nagata, and O. Akashi, "An efficient framework for data-plane verification with geometric windowing queries," *IEEE Trans. Netw. Service Manage.*, vol. 14, no. 4, pp. 1113–1127, Dec. 2017.
[43] H. Yang and S. S. Lam, "Collaborative verification of forward and reverse reachability in the Internet data plane," in *Proc. IEEE 22nd Int. Conf. Netw. Protocols*, Oct. 2014, pp. 320–331.
[44] G. Juniwal, N. Bjorner, R. Mahajan, S. Seshia, and G. Varghese, "Quantitative network analysis," Tech. Rep., 2016.
[45] K. G. Larsen, S. Schmid, and B. Xue, "WNetKAT: A weighted SDN programming and verification language," in *Proc. OPODIS*, 2016, pp. 1–28.
[46] H. Sharma, W. Wu, and B. Deng, "Symbolic execution for network functions with time-driven logic," in *Proc. 28th Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst. (MASCOTS)*, Nov. 2020, pp. 1–8.
[47] Z. Guo, J. Lin, Y. Bai, D. Kim, M. Swift, A. Akella, and M. Liu, "LogNIC: A high-level performance model for SmartNICs," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2023, pp. 916–929.
[48] S. Thomas, G. M. Voelker, and G. Porter, "CacheCloud: Towards speed-of-light datacenter communication," in *HotCloud*, 2018. [Online]. Available: https://www.usenix.org/conference/hotcloud18/presentation/thomas
[49] Z. Zhao and B. Wu, "Scalable SDN architecture with distributed placement of controllers for WAN," *Concurrency Computation, Pract. Exper.*, vol. 29, no. 16, pp. 1–26, Aug. 2017.
[50] C. Röver and T. Friede, "Discrete approximation of a mixture distribution via restricted divergence," *J. Comput. Graph. Statist.*, vol. 26, no. 1, pp. 217–222, Jan. 2017.
[51] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Rev.*, vol. 59, no. 1, pp. 65–98, Jan. 2017.
[52] K. Zhao, P. Goyal, M. Alizadeh, and T. E. Anderson, "Scalable tail latency estimation for data center networks," in *Proc. NSDI*, 2023, pp. 1–11.
[53] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The Internet topology zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011.
[54] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun.*, Aug. 2008, pp. 63–74.
[55] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon, "Shoal: A network architecture for disaggregated racks," in *Proc. NSDI*, 2019, pp. 1–19.
[56] A. V. Krishnamoorthy, H. D. Thacker, O. Torudbakken, S. Müller, A. Srinivasan, P. J. Decker, H. Opheim, J. E. Cunningham, I. Shubin, X. Zheng, M. Dignum, K. Raj, E. Rongved, and R. Penumatcha, "From chip to cloud: Optical interconnects in engineered systems," *J. Lightw. Technol.*, vol. 35, no. 15, pp. 3103–3115, Aug. 2, 2017.
[57] S. Yan, X. Wang, X. Zheng, Y. Xia, D. Liu, and W. Deng, "ACC: Automatic ECN tuning for high-speed datacenter networks," in *SIGCOMM*, 2021, pp. 1–15.
[58] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. K. Martin, M. McLaren, P. Chandra, R. Cauble, H. M. G. Wassel, B. Montazeri, S. L. Sabato, J. Scherpelz, and A. Vahdat, "1RMA: Re-envisioning remote memory access for multi-tenant datacenters," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun. Appl., Technol., Architectures, Protocols Comput. Commun.*, Jul. 2020, pp. 708–721.
[59] E. Kissel, M. Swany, B. Tierney, and E. Pouyoul, "Efficient wide area data transfer protocols for 100 gbps networks and beyond," in *Proc. 3rd Int. Workshop Network-Aware Data Manage.*, Nov. 2013, pp. 1–10.
[60] T. Yuan, X. Huang, M. Ma, and J. Yuan, "Balance-based SDN controller placement and assignment with minimum weight matching," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2018, pp. 1–6.
[61] I. Cho, K. Jang, and D. Han, "Credit-scheduled delay-bounded congestion control for datacenters," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2017, pp. 239–252.
[62] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. NSDI*, 2012, p. 20.
[63] C. Zhao, Y. Guo, J. Wang, Q. Qi, Z. Zhuang, H. Sun, L. Guo, Y. Xie, and J. Liao, "EPVerifier: Accelerating update storms verification with edge-predicate," in *Proc. NSDI*, 2024, pp. 1–29.
[64] J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "P-rex: Fast verification of MPLS networks with multiple link failures," in *Proc. 14th Int. Conf. Emerg. Netw. Experiments Technol.*, Dec. 2018, pp. 217–227.
[65] I. van Duijn, P. G. Jensen, J. S. Jensen, T. B. Krøgh, J. S. Madsen, S. Schmid, J. Srba, and M. T. Thorgersen, "Automata-theoretic approach to verification of MPLS networks under link failures," in *Proc. IEEE ToN*, 2021, p. 29.
[66] P. Zhang, D. Wang, and A. Gember-Jacobson, "Symbolic router execution," in *Proc. ACM SIGCOMM Conf.*, Aug. 2022, pp. 1–27.
[67] P. Namyar, B. Arzani, R. Beckett, S. Segarra, H. Raj, U. Krishnaswamy, R. Govindan, and S. Kandula, "Finding adversarial inputs for heuristics using multi-level optimization," in *Proc. NSDI*, 2024, p. 28.
[68] R. Li, Y. Yuan, F. Ye, M. Liu, R. Yang, Y. Yu, T. Guo, Q. Ma, X. Zeng, C. Xu, D. Cai, and E. Zhai, "A general and efficient approach to verifying traffic load properties under arbitrary k failures," in *Proc. ACM SIGCOMM Conf.*, Aug. 2024, pp. 228–243.
[69] Y. Chang, S. Rao, and M. Tawarmalani, "Robust validation of network designs under uncertain demands and failures," in *Proc. NSDI*, 2017, pp. 1–6.
[70] P. G. Jensen, D. Kristiansen, S. Schmid, M. K. Schou, B. C. Schrenk, and J. Srba, "AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks," in *Proc. 16th Int. Conf. Emerg. Netw. EXperiments Technol.*, Nov. 2020, pp. 1–18.
[71] N. Choi, L. Jagadeesan, Y. Jin, N. N. Mohanasamy, M. R. Rahman, K. Sabnani, and M. Thottan, "Run-time performance monitoring, verification, and healing of end-to-end services," in *Proc. IEEE Conf. Netw. Softwarization (NetSoft)*, Jun. 2019, pp. 30–35.
[72] A. H. Veen, "Dataflow machine architecture," in *Proc. CSUR*, 1986, p. 27.
[73] R. L. Smith, "Extreme value theory," in *Handbook of Applicable Mathematics*. Chichester, U.K.: Wiley, 1990.
[74] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *Acm sigplan notices*, vol. 2, pp. 1–22, 2014.
[75] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *Proc. SIGCOMM*, 2015, pp. 1–8.
[76] D. Basin, S. Krstić, and D. Traytel, "Almost event-rate independent monitoring of metric dynamic logic," in *Runtime Verification*, 2017.

SEPEHR ABDOUS received the B.Sc. degree in computer engineering from the Sharif University of Technology, in 2019, and the M.Eng. degree in computer science from Johns Hopkins University, in 2023, where he is currently pursuing the Ph.D. degree in computer science. His research interests include designing techniques to measure and address bursts, i.e., sudden spikes in networks load that results in packet drops and hurts performance, in large-scale data centers.

SENAPATI DIWANGKARA received the B.S. degree in informatics from Bandung Institute of Technology, in 2020. He is currently pursuing the Ph.D. degree in computer science with Johns Hopkins University. His research interests include assuring the reliability and security of both backend and frontend web technologies, such as networks and web frameworks.

SOUDEH GHORBANI received the B.Sc. degree in computer engineering from the Sharif University of Technology, in 2009, the M.Sc. degree in computer science from the University of Toronto, in 2011, and the Ph.D. degree in computer science from the University of Illinois Urbana–Champaign, in 2016. She is currently an Assistant Professor of computer science with Johns Hopkins University and her lab focuses on designing and building reliable and fast networking systems.

• • •