Probabilistic Network Latency Verification

Anonymous Author(s)

ABSTRACT

As modern networks evolved and performance need became more stringent, often expressed in terms of Service Level Agreement (SLA), there is a rising need to verify that these agreements hold true. Designing a network verifier that can practically reason about quantitative properties proves to be a unique challenge, since it needs to be able to expressively model many quantitative phenomenon, like congestion control and load balancing scheme, while being reasonably performant and not requiring too many inputs from its user. In this work, we introduce a network latency verification framework, Tempus, that could probablistically verify temporal properties of a network given latency measurements of its components. We also introduce two effective optimization techniques to exploit the symmetry in the verification process to significantly reduce the verification runtime. Our evaluation shows that Tempus can verify temporal property 90% faster on average than our baseline, which is devoid of all optimization. We also identify that highly symmetrical topology, like Fat Tree, benefits from our optimization the most and can cut the amount of convolution up to 6 orders of magnitude.

1 INTRODUCTION

Many modern network deployments often necessitate tight performance guarantee in order to sustain their increasing demand. At the same time, end users also have a rising performance standard with regards to their internet browsing experience, forcing companies to optimize every stack of their deployment in order to increase conversion. Hence, network operator must find a way to verify that the network that they configure meets this demand, or accept the reality of Murphy's Law.

Over the last decade, there has been a lot of development in the area of network verification generally [1][2]. Still, most of these verifiers are primarily fixated on verifying *qualitative* properties – the convergent functional behavior of a network under various failure scenarios – such as reachability and loop detection, and not *quantitative* properties related to performance, such as bandwidth and latency. Compared to the verification of reachability-based property, verification of bandwidth and latency is inherently more challenging since it introduces more dimension to the problem. In addition to reasoning about the existence of a flow, one needs to reason about the measured detail of said flow and model their interaction.

There has been a few notable works that push network verification concept towards this direction [3] [**?**]. These newer tools model a wide range of additional network behavior that might affect performance to answer a specific question in mind, namely link-load violation and congestion control behavior. However, their approaches so far has been focused on worst-case analysis, modeling and answering whether an unwanted event could happen or not, which requires further statistical interpretation in order to translate them to a percentage guarantee that appear on an SLA.

Moreover, none of these tools were designed to answer questions related to end-to-end latency SLA [4]. Ideally, such a tool would answer the challenge of modeling various latency altering network behavior, such as traffic generation, load balancing scheme, and congestion control, and expresses the important statistical property of the resulting end-to-end latency. Not just in terms of worst-case behaviors and tails, but also the means.

In this work, we propose a verification framework to probabilistically verify the latency property of packets traversing from a source to destination node under various failure scenarios, by using latency measurements of the components in the network.

We introduce the design and implementation of Tempus, a probabilistic network latency verification framework. Tempuswill use the latency information from relevant component measurements (e.g. router queue length) to infer the latency distribution of said component. Assuming that future traffic is i.i.d., we then employ a numerical convolution method to combine the latency distribution of each relevant components together to produce the end-to-end latency distribution of an src-dst router pair.

To determine said relevant components, we built Tempuson top of an existing qualitative verification framework to efficiently explore the path used for an src-dst router pair given various scenarios of failures. By obtaining the end-to-end latency distribution from this information and the latency measurement, we could analyze the statistical properties of the src-dst pair latency distribution to probabilistically argue about the latency properties.

We also introduced two optimization techniques on top of this framework to reduce the verification time by multiple orders of magnitude. These two optimization techniques rely on the symmetry that we have found in the quantitative verifier when brought into the context of latency verification.

Our evaluation shows that the verification framework, with the help of our optimization techniques, could accomplish the verification task on various WAN and datacenter topologies in the order of minutes?

With this work, we make the following technical contributions:

- **Novel temporal verification framework** using numerical convolution of network components' latency measurement.
- **Two optimization techniques** in said verification framework by exploiting symmetry in the failure exploration.
- **Tempus**, an implementation of our verification framework and its optimization on Julia.

The code for this work is open-sourced at link. This work does not raise any ethical issues.

2 LATENCY MODELING BACKGROUND

In the context of verification, there are multiple approaches to model latency in a network. We will briefly compare 4 possible methods that might be useful to attack this problem and and its limitation to illustrate the methodological challenge to accurately model latency.

2.1 Simulation

One of the available methods to reason about latency in a network is by simulating it using a network simulator (e.g. ns3, omnet). On a high level, the tool will simulate the lifecycle of packets, created by a load generator, and the user could fail some network components and collect the packets' latency measurement to see how the perturbation affects the latency property in question.

Network simulator has been widely used in our community as an alternative to experimenting on a bare-metal system. While generally accurate, network simulators notoriously take a long time to run. Moreover, the simulation workflow is more akin to testing than verification, relying on the user to create failure scenarios and reason about latency in a contrapositive way – the absence of a negative result does not mean the property is guaranteed to be fulfilled.

2.2 Timed Automata

Timed Automata is an extension to the classic Finite State Machine (FSM) in automata theory that adds a global clock to be used as transition constraints. The runtime will keep a set of global clock that monotonously increases and can be individually reset. Using these clocks, we could construct the state machine that constraints each transition with a time limit. With this additional time constraints, we could verify properties in a similar way as traditional model checking methods (e.g. using CTL formulas).

While this technique offers a proper verification framework and a higher level of abstraction compared to simulation, traditional Timed Automata theory is deterministic, offering no random transitions to express random component failure and a way to reason about latency probabilistically.

A popular Timed Automata modeling tool, UPPAAL, tried to alleviate this determinism problem by introducing Statistical Model Checking (SMC) on top of their Timed Automata model. The feature allows user to introduce random transition (with configurable weights) and distribution based transition, in order to probabilistically reason about the desired property and model random failures of network components.

However, they did it by collecting the statistical property of multiple simulation runs, which has the same drawback as the simulation-based method. It is also limited in its choice of probability distribution, only allowing user to use uniform and exponential distribution.

2.3 Queuing Theory

Another theoretical framework that is more closely related to networking devices are queuing theory. Queuing theory allows you to reason about the equilibrium latency of a given traffic, given the arrival and processing distribution to each component, among other things.

One major downside of this framework, however, is the limitation of assumed probability distributions that is being used in each queue and the arrival process. Where one could derive a closed-form solution for a network where the queues and / or the arrival process are identified as a poisson process (i.e. Jackson network, BCMP network [5]), the same thing cannot be said for a queuing network that posits a more diverse queuing behavior. A framework to describe a queuing network with an arbitrary queuing behavior and arrival process in this theory is yet to be discovered.

2.4 Tempus

Considering these available approaches and their respective strength and limitation, we devise a simple yet powerful framework to verify the latency property in a given network.

Our main idea is to model the latency of a given network component as an independent random variable. By modeling the latency this way, we could then operate on those random variable to model the latency of a given path, and eventually the latency of a given src-dst pair. Using the resulting random variable, we could then determine the temporal property in question by analyzing the statistical property of the resulting probability distribution.

This framework is expressive. The packet interarrival time that is represented by the latency distribution in each router encodes the latency information of the various traffic

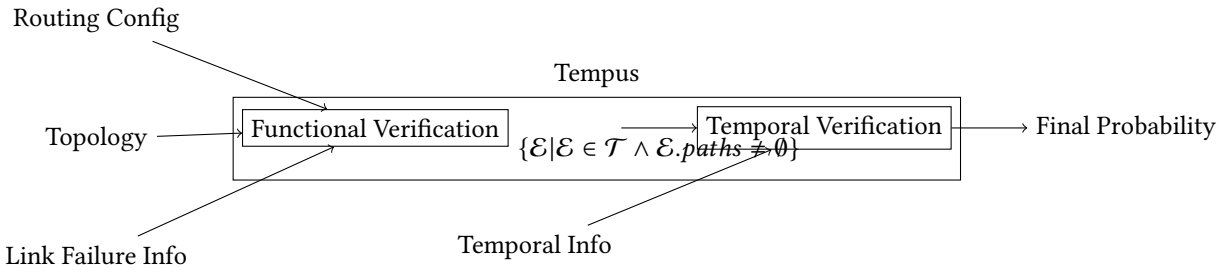

Figure 1: Overview of Tempus

that is going through the network. At the same time, it also encodes the effect of congestion and flow control scheme that affects those traffic.

This framework is performant. Operating on the high-level of abstraction of the latency distribution in each network components allows this framework to perform the verification task in a matter of minutes? We demonstrate our evaluation result on Section 6.

This framework is practical. Queue length is a common metric that is used to measure the resource utilization of a network [6] [7]. Using this metric, one could easily construct a latency distribution by combining it with the line rate bandwidth.

We divide the problem of latency verification in this framework into two parts: verifying that two nodes are reachable (**functional verification**) and only if the functional property is fulfilled, we would verify whether the latency between two nodes fulfilled some temporal condition (**temporal verification**). In Section 3, we briefly describe the functional verification scheme that our framework is built upon. Next, in Section 4, we describe our novel temporal verification framework in more detail. Next, in Section 5, we describe the optimization techniques that arise from these two steps. We then evaluate our implementation in Section 6. Finally, we will touch on some related works on Section 8, limitations and future works on 7, and conclude our work with Section 9.

3 FUNCTIONAL VERIFICATION

For functional verification stage, NetDice [8] has laid the way for verifying reachability between two nodes under failure in a probabilistic and efficient manner. To contextualize our modification, we will briefly describe some parts of NetDice's network encoding and exploration algorithm as a prelude to our alteration. For a more complete complete description of NetDice, please refer to the original paper.

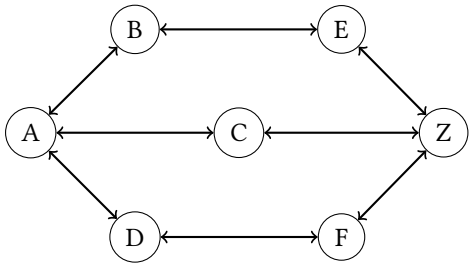

Figure 2: Example topology, we want to check the reachability of A to Z

3.1 Topology Graph

In this framework, a network is encoded in an edge-labeled directed graph $G_t = (V_t, E_t)$ where V_t represents the nodes in the network and E_t represents the functional connectivity between a source and a destination node. A physical link is then represented as a pair of symmetrical edges that shares the same source and destination node but with opposite direction (e.g. $v_1 \rightarrow v_2$ and $v_2 \rightarrow v_1$). A routing protocol is then defined on top of this graph by some auxilary information.

To explain it with further clarity, we use the topology in Fig. 2 as a running example for this paper. We will assume that the nodes in the network runs OSPF with weight 1 for every edge and ECMP as their load-balancing scheme. We want to check the temporal probability of packets that departs from A to Z.

To represent random failure, NetDice defined a universal link failure rate (i.e. the probability of *any* link in the network randomly failing). We refine NetDice's model slightly by allowing each links to have different failure rate. We label each edge in E_t with a function $r : E_t \rightarrow \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ that represents the failure rate of a given physical link. As a consequence, two symmetrical edges (i.e. two edges that shares the same node pairs but with opposite direction) will also share the same failure rate, and will be disabled in a coupled fashion.

For the sake of example, in our Fig. 2 topology, we will assume that each link has a 10% chance of failure.

3.2 State Exploration

A **network state** is defined as a set of failed links. There are $2^{|E_t|}$ network states in a given network and a brute-force strategy would need to explore all of the states to compute the reachability property of a node pair in a given network. NetDice however, will merge multiple network states into an **equivalence class** by marginalizing over *cold edges*, links whose failure is guaranteed not to change the convergent path of a highlighted network state.

NetDice will systematically explore many equivalence classes in the network. It will start from the equivalence class of a perfect network and will fail certain *hot edges* (i.e. edges that are not cold) to explore another equivalence class. While not explicit in its description, NetDice's algorithm will effectively form an **exploration tree**.

In order to preserve some information that will be used in the temporal verification stage, we modify this original exploration algorithm to make it more explicit. We started by formalizing the notion of Equivalence Classes. An Equivalence Class $\mathcal{E}$ is defined as a 3-tuple $(U_h, D_h, paths)$. U_h and D_h refers to a set of hot edges that are up and down respectively. $paths$ refers to the convergent paths between src-dst pair that is produced by the control plane when the links in D_h fails.

We then define the Exploration Tree $\mathcal{T}$ as a tree of Equivalence Classes. At the root, we have an Equivalence Class that corresponds to a perfect network, where $paths$ is the path(s) that the control plane will produce given a perfect network condition and $U_h = D_h = \emptyset$. Each Equivalence Class in this tree will have children which have one of its parent's link in $paths$ appended to its D_h, essentially failing one of the link in the path. If an Equivalence Class has an empty $paths$, then it would have no children.

To compute the functional probability P_f of a given Equivalence Class, we compute the product of the 'up' probability of all the links in U_h and $paths$ and the product of the 'down' probability of all the links in D_h.

In our running example, we could see the subset of the exploration tree in Fig. 3. We see that in a perfect network ($\mathcal{E}_1$), OSPF will produce ACZ as the shortest path between A to Z. The probability of this Equivalence Class actually materializing (P_f) is 0.81, which is the probability of AC and CZ being up at the same time (0.9^2).

In the Equivalence Class where the link AC is failing however ($\mathcal{E}_2$), OSPF and ECMP will produce two equaly-weighted paths $ABEZ$ and $ADFZ$. The probability of this Equivalence Class actually materializing (P_f) is 0.053 ($0.9^6 \cdot 0.1$).

After each equivalence class produced its convergent paths, we then continue to verify the temporal property of that equivalence class in the next stage.

4 TEMPORAL VERIFICATION

In the temporal verification stage, we will use the path information that we got from functional verification stage and augment them with latency information to produce the final temporal property probability.

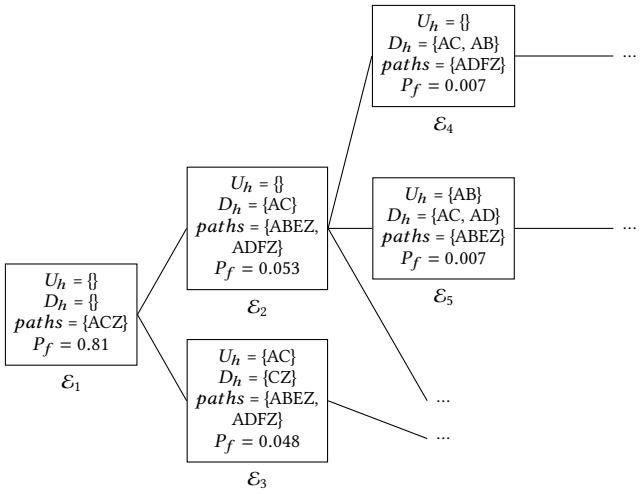

Figure 3: Subset of Modified Exploration Tree $\mathcal{T}$ for our example topology, $\mathcal{E}_2$ and $\mathcal{E}_3$ will get consolidated

4.1 Latency Label

Similar to link failure r, we use edge-labeling technique to represent latency that will get introduced by the connectivity between two nodes. Our model will model two significant sources of latency: **link propagation** and **packet queuing** in the node. We will encode these latencies by equipping the model with two additional labels.

Encoding link propagation latency is fairly straightforward. We define a function $l_p : E_t \rightarrow \mathcal{D}$ where $\mathcal{D}$ is a set of continuous univariate distribution that has a minimum value of 0. This distribution signifies the time it would take for the respective physical link to transmit a packet from one end to another. Because of this, just like r, two symmetrical edges will share the same distribution.

To encode queuing latency, we first note that the queuing mechanism in modern switches usually resides in the output port. Since one port in a switch is only connected to one port in another switch, we could effectively assign queuing latency to the connectivity between switches. To do this, we define another function $l_q : E_t \rightarrow \mathcal{D}$. This distribution signifies the output queue latency in the source node that

is going to be forwarded to the destination node. Unlike l_p, two symmetrical edges will have two different distributions since they represent two different output queues.

4.2 Latency Measurement as Distribution

Given *paths*, l_p, and l_q, we could then compute P_t, the probability of a given temporal property being fulfilled. We will describe our algorithm for computing this probability in the next few sections.

4.3 Weighted Average and Path Convolution

Since a convergent behavior of the forwarding plane *paths* might contain more than one paths, we will compute the temporal probability of *paths* by computing the weighted average of the temporal probability of each of the individual path. The specific weight depends on the load-balancing method used. In our example, we will use ECMP load-balancing scheme.

To get the temporal probability of a single path, we will use the latency information from l_p and l_q of each edge in the path to compute the latency distribution of the whole path. To do this, we resort to the methods of **convolution**. For each edge e in the path, we will get $l_p(e)$ and $l_q(e)$ and convolve them all together to get the path latency distribution.

With this path latency distribution, we could get the temporal probability of a path by computing the statistical property of said distribution. In this work, given a path latency distribution $\mathcal{L}$, we define two temporal properties:

- **Bounded Reachability**: the probability that a packet will get transmited below t time unit. This will get computed as $cdf(\mathcal{L}, t)$
- **Tail Reachability**: the probability that a packet will get transmited above t time unit. This will get computed as $1 - cdf(\mathcal{L}, t)$

In our Fig. 3 Exploration Tree for example, $\mathcal{E}_2$ has two equally probable path, *ABEZ* and *ADFZ*. We will first do a chain convolution of all the distributions in *ABEZ* ($l_p(AB)$, $l_q(AB)$, ..., $l_q(EZ)$) and do the same thing for *ADFZ*. We will then take the *CDF* of $\mathcal{L}_{ABEZ}$ and $\mathcal{L}_{ADFZ}$ with our desired t and average them out to get P_t of $\mathcal{E}_2$.

In short, given *paths*, l_p, and l_q, we will do the following:

(1) Split *paths* into its individual path
(2) For each path, compute its latency distribution by convolving the latency distribution l_p and l_q of each link
(3) With the resulting path latency distribution, determine the probability of temporal property by computing the statistical property of said distribution
(4) Combine the temporal property of each path by computing the weighted average, based on the load-balancing scheme of the control plane

We do each of these steps to every Equivalence Class in $\mathcal{T}$, and do a sum-product operation of P_f and P_t to get the final probability of the property in question.

At this point, by introducing the temporal verification algorithm, we're essentially adding additional overhead to the exploration algorithm (on top of functional verification) that scales to the size of the Exploration Tree. In other words, for $\mathcal{T}$ of size n, we will need to do temporal verification in each of those n Equivalence Class. We aim to minimize the overhead of temporal verification by introducing some optimization algorithms later.

4.4 Numeric Convolution

There is one major problem with a convolution-based technique for computing the latency distribution of a given path: not every distribution pair can be convolved analytically. A closed-form solution of a convolution is usually only available for two distributions of the same type.

In our example, we define $l_q(EZ)$ to be lognormal distribution, which doesn't have a closed form convolution solution with the other latency distribution in the network, which is of gamma distribution form. Therefore, in order to convolve two arbitrary distributions, we need to leverage a numerical convolution method.

We leverage an existing numerical convolution algorithm, DIRECT, to fulfill this role. We choose this method due to their bounded error property: DIRECT guarantee that the computed distribution and the correct theoretical distribution has a KL-divergence below a certain bound. DIRECT has also been implemented in R's popular bayesmeta package.

One subtle detail about DIRECT is while convolution is defined to be a commutative operation, and DIRECT also achieves this property, the order of operation matters for the algorithm's runtime performance. In particular, if we had a chain of DIRECT convolutions, the result of a convolution should not be set as the second input distribution in the later convolutions to avoid performance penalty. This is caused by a nested loop from the following two mechanisms:

- DIRECT works by computing a list of support values by doing many PDF queries of the second input distribution
- Computing the PDF of a DIRECT distribution (the returned distribution of a DIRECT algorithm) itself involves iterating over the current support values

As an ilustration, if we had 3 random variables A, B, and C, and we want to convolve them all together, it would be faster to compute $direct(direct(A, B), C)$ instead of $direct(C, direct(A, B))$

While this problem is trivial to solve (swap the input parameter if the latter is detected as a DIRECT distribution), it does limit the type of optimization that we could have in the chain convolution process.

5 OPTIMIZATION

5.1 Equivalence Class Consolidation

To minimize the amount of temporal verification overhead, we want to find some symmetry in the existing exploration algorithm so that we could reduce the exploration space even further.

In the original NetDice exploration algorithm, multiple network states were merged into an Equivalence Class based on its cold edges (edges whose failure won't change the convergent path(s)). In other words, network states within the same Equivalence Class will have the same convergent path(s).

However, we observe that each of those Equivalence Classes are not *unique*: while network states within the same equivalence class shares a convergent path(s), **multiple equivalence classes could also share the exact same convergent path(s)**, as shown in $\mathcal{E}_2$ and $\mathcal{E}_3$ in Fig. 3.

Since we define latency distribution to marginalize over all factors other than the path, we could effectively *consolidate* these equivalence classes into one big equivalence class and do temporal verification once.

We implemented this idea by doing memoization on the temporal probability of a given convergent path(s). We will only do temporal probability computation once, when we first iterate over an equivalence class that has a certain convergent path(s), and we cached the result should another equivalence class with the same convergent path(s) emerged.

5.2 Path Memoization

After consolidating many equivalence classes that shares the same convergent path(s), we are left with fewer, bigger, but unique equivalence classes. As significant as it is, we could reduce the computation cost further by drawing connections between these unique equivalence classes.

In a network with a load-balancing protocol, the convergent state of the data plane might forward a packet through multiple possible paths. In the context of our framework, we say that an equivalence class might have more than one convergent paths. Nevertheless, we note that two equivalence classes with different convergent paths might not be two independent subset. In other words, **multiple equivalence classes could share a subset of individual path**.

Since the way we compute convergent path(s) temporal probability is by calculating the weighted average of individual path temporal probability, we could reduce the amount of convolution we will need to do by doing further memoization on the temporal probability of a given path. If we were to compute the temporal probability of an equivalence class with previously cached individual path, we only need to calculate the weights that corresponds to the load balancing protocol without doing any convolution.

In our running example, we could see that in Fig. 3, $\mathcal{E}_2$ and $\mathcal{E}_4$ share an identical path, $ADFZ$. We could therefore memoize $cdf(\mathcal{L}_{ADFZ}, t)$ when we explore $\mathcal{E}_2$ and query the cached result when we explore $\mathcal{E}_4$.

With this optimization, we will do a chain convolution by the amount of path available in the network between the source and destination (which is 3 in the case of our running example in Fig. 2: ACZ, $ABEF$, and $ADFZ$).

6 EVALUATION

We implemented Tempus in $\sim$ 600 lines of Julia [9] code. This code will take a topology configuration, source and destination node, and the temporal property in question and its threshold. It will then output the final probability of the network upholding that temporal property. We ran our prototype on a machine with 2 Intel E5-2630 v3 8-core CPU running up to 2.4 GHz and 128 GB of ECC RAM, provided by CloudLab [10].

To appraise the viability of our approach, we seek to answer 5 evaluation questions:

(1) How long does it take to run the verifier overall? How does it scale?
(2) What is the bottleneck step in the verification process?
(3) How effective is the optimization technique in reducing the verification time?
(4) How effective is the optimization technique in reducing the explored equivalence classes?
(5) How does different topologies affect optimization effectiveness?

6.1 Experiment Setup

Topology	#routers	#links	type
Latnet	69	74	WAN
Highwinds	18	31	WAN
AT&T	25	56	WAN
Uninett	74	101	WAN
GtsCe	149	193	WAN
Fat Tree 4	20	32	DC
Fat Tree 6	45	108	DC
Fat Tree 8	80	256	DC
Fat Tree 10	125	500	DC

Table 1: List of topologies being tested

We tested Tempus on 5 real-world WAN topologies from the Topology Zoo and 3 datacenter topologies (fat-tree, with various amount of pods).

6.1.1 Functional Verification. As verifying the routing protocol functional behavior is not the primary contribution of our research, we simply use OSPF as the routing protocol in our experiment with uniform link weights of 1. We set 0.1% failure rate in all links, in accordance to the failure rate in previous studies. We then run the functional verification with 10^{-8} inaccuracy level.

In order for our evaluation to be realistic, we set the source and destination node to be one of the edge routers (node with the smallest degree / smallest centrality?).

6.1.2 Temporal Verification. For latency distribution, we show that we can use an empirical measurement as a distribution by using the reported queue length from DCTCP [6]. By multiplying the queue length distribution with the line-rate, we could approximate the queuing latency distribution for a given router.

Since the CDF operation that we define for temporal property verification is relatively cheap, we expect our result to generalize with any threshold we chose. In other words, while changing the threshold will change the final probability, it won't change the runtime performance of Tempus. Thus, we set an arbitrary threshold for our evaluation.

6.2 Runtime Profiling

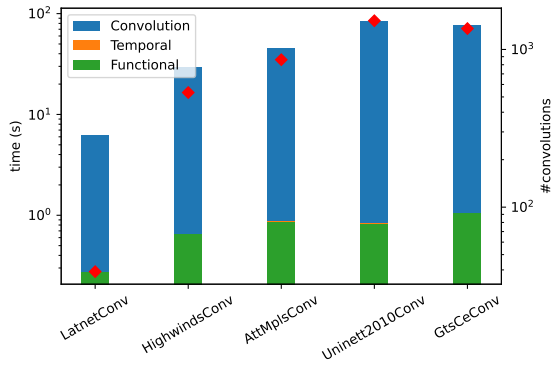

Figure 4: Runtime performance of the verification process, red dots represent the number of convolutions

6.2.1 Performance and Scalability. To begin our evaluation, we first measured the running duration of the verification process (and its steps) on various topologies. Our results in Fig. 4 shows that our verification technique finished within a reasonable timeframe. The verifier finished in the order of minutes, even for topologies with hundred of nodes.

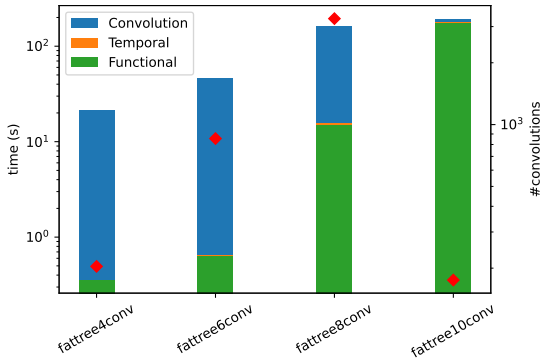

Figure 5: Fat Tree, red dots represent the number of convolutions

From the same result, we also note that our verifier scales gracefully over the size of the network. The verification duration ranges from 6 seconds to around 15 minutes. To drive this point more precisely, we also evaluate the running time of fat-tree topology in various sizes. Our results in Fig. 5 shows that by keeping the same type of topology and scaling them up, we increased our verification time almost linearly, while the unoptimized version timed out after 2 hours. Not only that, our combined optimization strategies actually reduces the time for fat tree topology of $k = 10$ compared to equivalent topologies with smaller size.

6.2.2 Bottleneck. By looking at the proportion of time spent on each step in Fig. 4, we could see that the combined **convolution procedure is the bottleneck** step in our verifier. The convolution procedure takes 95% - 99% of the duration of the overall verification. Hence we can see in the same figure that the runtime performance of Tempus and the total amount of convolution is highly correlated.

We conclude that for a reasonably low imprecision level, **Tempus runs in the order of minutes** and the performance of Tempus is primarily **bottlenecked by the total convolution operations**.

6.3 Optimization Effectiveness

We have established that the convolution procedure is a relatively expensive operation within our verifier. Next, we inspected the effectiveness of our optimization techniques in reducing them. To do that, we measured the overall running duration of the verifier while selectively disabling the optimization.

Our results in Fig. 6 shows the effect of the two optimization techniques we introduced – consolidation and memoization – in reducing the verification duration and amount of convolution (both in log scale). For baseline, we ran the verifier without any optimization, resulting in the left bar.

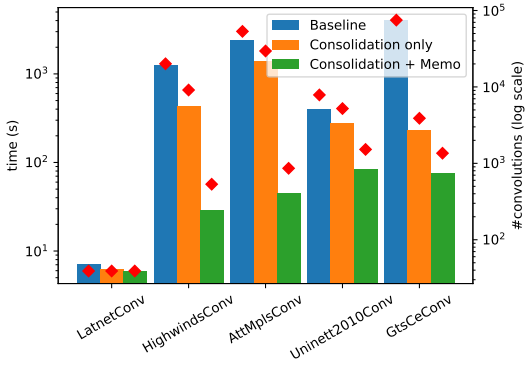

Figure 6: Amount of convolution (in log scale) depending on what optimization strategy is applied

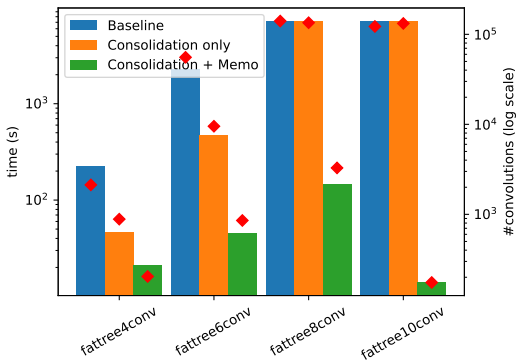

Figure 7: Amount of convolution (in log scale) depending on what optimization strategy is applied

The middle bar represents the result where we enable only the consolidation strategy. Finally, the right bar represents the result where we enable both the consolidation and memoization strategy.

We note that for most of the topologies, the combination of both optimization strategies resulted in **79% - 98% improvement in performance**. Compared to the baseline performance, the consolidation strategy contributes to 30% - 94% of the improvement and the memoization strategy contributes to 67% - 96% on top of that.

We conclude that **consolidation and memoization are both effective optimization strategies** in our verification framework.

6.4 Equivalence Class Reduction

While we cannot directly compare Tempus with other verifiers (due to difference in verification goal), we could use the amount of equivalence class as an indirect proxy of the verifier's behavior and performance. Therefore, we measured

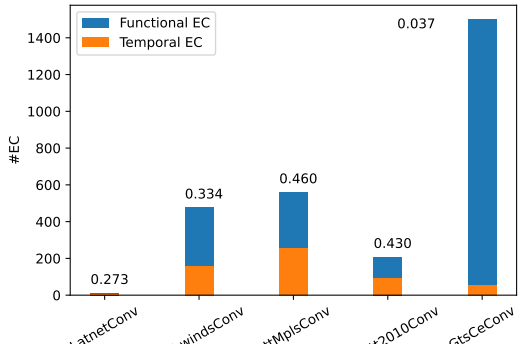

Figure 8: Amount of temporal EC compared to functional EC, label represents ratio

the amount of additional equivalence classes introduced by Tempus in order to indirectly compare its overhead in addition to its functional counterparts. This result is only influenced by the consolidation strategy, since the memoization strategy operates on a smaller granularity than equivalence classes.

Our results in Fig. 8 shows the ratio between the amount of temporal equivalence classes that is being re-explored and the amount of functional equivalence class. We note that we only need to **re-explore 4% to 46% equivalence classes** as an additional overhead compared to functional verification.

We conclude that **consolidation is effective in reducing the amount of equivalence classes that needs to be re-explored** in our verification framework.

6.5 Optimization Analysis

Top.	# path (opt)	# path (cons)	# path (base)	# conv (opt)
ft4	20	92	251	204
ft6	81	1,278	7,857	855
ft8	304	42,892	500,195	3280
ft10	25	456,700	6,669,169	175
ft12	36	14,367,491	288,537,289	252
ft14*	49	23,767,482	323,686,008	343

Table 2: Effectiveness metric

Up to this point, we have demonstrated that consolidation and memoization are two effective strategies in reducing the verification duration and amount of convolution procedure that needs to be done. However, the results also show that their effectiveness varies depending on the topology.

6.5.1 Why is Fat Tree 10 better than Fat Tree 4? To analyze what affects the effectiveness of our optimization technique,

we will start by explaining one peculiar result in our evaluation: despite having a bigger size, the temporal verification runtime of fat tree with $k = 10$ is faster than $k = 4$.

This is due to the fact that the shortest path in $k = 10$ is a lot more diverse than that of $k = 4$. In a 3-tier Fat Tree topology, an edge node could reach another pod's edge node in at least 4 hops. The amount of this shortest path is $(k/2)^2$, which is the same as the number of core node.

We could see from Table 2 that when $k \geq 10$, the amount of unique path explored by the functional verifier is exactly the same as the number of core node. Combined with the fact that the average amount of convolution per path is 7 (which suggests that the path length is 4), means that the functional verifier only produces equivalence classes that consisted of only different combination of these shortest paths.

When $k < 10$ however, the functional verifier also needs to produce equivalence classes that results in a longer convergent paths in order to reach the same accuracy level. in Fat Tree, this is usually marked by a visit to another aggregation node in another pod. We could see from Table 2 that when $k < 10$, the amount of of convolution per path is more than 7 (which suggests that a path longer than 4 hops exist).

6.5.2 Generalization. From the insight of this particular result, we could draw two properties in a general network that could determine the runtime of our verifier.

The first one is the **amount of possible paths**. If the amount of possible paths between a src-dst pair in a given topology is small, then the functional verification step would produce fewer equivalence classes and end early. While most of our evaluated topology had a lot of possible paths, one exception to this is LatNet, which only have 3 possible paths.

The second is whether the src-dst pair in a given topology would **produce paths that are "symmetric"**. By symmetric, we mean that the routing and load balancing method produce a lot of identical equivalence class and / or equivalence class that share the same path. As our explanation about Fat Tree 10 suggests, larger Fat Tree has a lot of path with the same length. This would result in a lot of symmetrical equivalence classes under ECMP. Our optimization techniques could identify this symmetry and reduce the amount of convolution in the temporal verification step.

7 LIMITATION AND FUTURE WORKS

Addressing Distribution Independence The convolution of two probability distributions assumes that the two distributions are *independent* of each other. While we accept this as a limitation of our model, latency in real networks (particularly the latency of two queues in the network) might be dependent of each other.

Queuing theory has laid some ground work to describe the asymptotic behavior of such queuing network. However, they assume some information about the incoming traffic and can only describe a network with a particular traffic or delay distribution, like poisson.

Thus, eliminating the independence assumption without demanding an unwieldy amount of information from the user while maintaining (or even improving) the accuracy of the model remains an open challenge and an interesting future direction.

Optimizing Functional Verification In our verifier, we built our temporal verifier on top of an existing functional verifier design. While it makes the design of the verifier simpler, since there is clear separation of concern, there might be some performance benefit to gain if we integrate these two parts further.

While we have proposed and evaluated optimization techniques for the temporal verification step, integrating this optimization to the functional verification step – perhaps by consolidating the equivalence classes before it is brought to the temporal verification stage in some way – might make our approach more scalable.

While our evaluation shows that the functional verification stage is not the bottleneck of our design, discovering the symmetry in the exploration state that might accomplish this task might bring an interesting insight to the network verification community in general.

8 RELATED WORKS

Deterministic Data and Control Plane Verifier There has been a substantial body of work concerning the functional behavior of routers with a given Forwarding Information Base (FIB) or Routing Information Base (RIB).

Prior works like HSA [1], NetKAT [11], and VeriFlow [2] focuses on verifying the current data plane forwarding behavior against some property. While useful in some real-time cases, these tools are not built to explore the many possible forwarding table given an enumeration of failure cases.

Control plane verifiers such as ARC [12], Minesweeper [13], and Tiramisu [14] focuses on verifying the various data plane states that could be produced by a control plane configuration over multiple failure scenarios. Many researchers has built on top of this idea, such as DNA [15] where they focuses on verifying the change in property in the event of configuration update.

As comprehensive as it is, both approaches only considers the functional behavior of the network (e.g. reachability, loop detection) and they traditionally verify the stated properties in a black or white fashion, only determining whether a given property is violated or not.

As a result, unlike Tempus, its usefulness does not extend into the use cases of verifying SLA, where such binary guarantees are often rightfully avoided in favor of a probabilistic and/or quantitative agreement.

Probabilistic Verifier More recent works have tried to address this very issue. Works like ProbNetKAT [16] (a probabilistic verifier based on NetKAT [11]), NetDice [8], and SRE [17] provides the user ability to define probabilistic failure model, where a given component of the network can fail with a given probability.

However, most of them still considers the functional property of the network as the main focus. While they are useful in verifying certain types of SLA (e.g. availability), they don't focus on more quantitative aspect of the network (e.g. bandwidth, latency).

Quantitative Verifier Some recent works are an exception to this trend, however. Works like QARC [3] (a quantitative verifier based on ARC [12]) has developed a framework to verify one particular quantitative property: excess bandwidth on a certain link. While QARC is a deterministic verifier, like Tempus, they focuses on a more quantitative property. Unlike Tempus however, this work is orthogonal to verifying latency and thus complementary to Tempus.

Network Calculus CCAC?

9 CONCLUSION

- Tempus is a scalable latency SLA verifier. It could probabilistically verify in the order of minutes
- Consolidation and memoization is an effective optimization strategy, although the exact magnitude of its effectiveness will depend on the network topology, specifically on the amount of paths and their length.

REFERENCES

[1] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 113–126, 2012.

[2] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the first workshop on Hot topics in software defined networks*, pp. 49–54, 2012.

[3] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella, "Detecting network load violations for distributed control planes," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 974–988, 2020.

[4] "Verizon global latency and packet delivery sla." https://www.verizon.com/business/terms/global_latency_sla/. Accessed: 2022-09-20.

[5] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *Journal of the ACM (JACM)*, vol. 22, no. 2, pp. 248–260, 1975.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 63–74, 2010.

[7] G. Kumar, N. Dukkipati, K. Jang, H. M. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, *et al.*, "Swift: Delay is simple and effective for congestion control in the datacenter," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pp. 514–528, 2020.

[8] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev, "Probabilistic verification of network configurations," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pp. 750–764, 2020.

[9] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[10] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, pp. 1–14, July 2019.

[11] 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. 49, no. 1, pp. 113–126, 2014.

[12] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 300–313, 2016.

[13] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 155–168, 2017.

[14] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 201–219, 2020.

[15] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li, "Differential network analysis," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 601–615, 2022.

[16] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic netkat," in *European Symposium on Programming*, pp. 282–309, Springer, 2016.

[17] P. Zhang, D. Wang, and A. Gember-Jacobson, "Symbolic router execution," in *Proceedings of the ACM SIGCOMM 2022 Conference*, pp. 336–349, 2022.