Mitigating Scalability Walls of RDMA-based Container Networks

Wei Liu[1,2], Kun Qian[2], Zhenhua Li[1✉], Feng Qian[3], Tianyin Xu[4], Yunhao Liu[1]
Yu Guan[2], Shuhong Zhu[2], Hongfei Xu[2], Lanlan Xi[2], Chao Qin[2], Ennan Zhai[2]
[1]*Tsinghua University* [2]*Alibaba Cloud* [3]*University of Southern California* [4]*UIUC*

Abstract

As a state-of-the-art technique, RDMA-offloaded container networks (RCNs) can provide high-performance data communications among containers. Nevertheless, this seems to be subject to the RCN scale—when there are millions of containers simultaneously running in a data center, the performance decreases sharply and unexpectedly. In particular, we observe that most performance issues are related to RDMA NICs (RNICs), whose design and implementation defects might constitute the "scalability wall" of the RCN. To validate the conjecture, however, we are challenged by the limited visibility into the internals of today's RNICs. To address the dilemma, a more pragmatic approach is to *infer* the most likely causes of the performance issues according to the common abstractions of an RNIC's components and functionalities.

Specifically, we conduct *combinatorial causal testing* to efficiently reason about an RNIC's architecture model, effectively approximate its performance model, and thereby proactively optimize the NF (network function) offloading schedule. We embody the design into a practical system dubbed ScalaCN. Evaluation on production workloads shows that the end-to-end network bandwidth increases by 1.4× and the packet forwarding latency decreases by 31%, after resolving 82% of the causes inferred by ScalaCN. We report the performance issues of RNICs and the most likely causes to relevant vendors, all of which have been encouragingly confirmed; we are now closely working with the vendors to fix them.

1 Introduction

Containers have become a pivotal technology in cloud computing and serverless computing [34, 35, 56, 70], which allow developers and service operators to deploy applications or functions onto the cloud rapidly and conveniently. Different from a heavyweight and full-fledged virtual machine, a container is merely a lightweight and standalone software bundle that includes all the code and dependencies needed by a specific application [30, 43]. The portability, lightweight, and isolation characteristics make containers an ideal choice for running a variety of tasks in the cloud, such as training foundation models [27, 31, 64] and building microservices [51, 53].

Container networks enable containers to communicate with each other seamlessly based on their hosts. To achieve high-performance data communications, almost all of today's container service providers (CSPs) utilize RDMA NICs (RNICs) to offload data traffic and network functions (NFs) from the host OS to dedicated hardware (RNICs) [35, 57]. The resulting RDMA-offloaded container network (RCN) is reported to have at least three times of performance gain compared to the traditional network stack inside the host OS [35, 39, 40], paving an efficient and generic network runtime for cloud applications and stateful/statelss functions.

Nevertheless, as a mainstream CSP, we observe that the performance gain of the RCN seems to be subject to its scale. During our two-year (03/2021 – 03/2023) operations on a production RCN (which involves ∼8K hosts equipped with ∼40K RNICs, serving 0.5M active containers simultaneously on average and ∼1M containers at peak hours), we notice that the end-to-end performance in terms of bandwidth and latency often decreases sharply and unexpectedly. For instance, when the number of active containers grows from 0.4M to 0.8M, the end-to-end bandwidth can decrease by 87%, and the packet forwarding latency can increase by 34×. In other words, there are "scalability walls" in the production RCN, which is rarely understood as existing studies mostly examine the RCN in small-scale and controlled settings [35, 58].

From Phenomena to Symptoms. To deeper understand the problem, we built a continuous monitoring infrastructure on top of our production RCN. It gathers cross-layer runtime data on each host, such as container resource consumptions, virtual switch flow tables, kernel logs, and RNIC statistics, during our monitoring period from 04/15/2023 to 04/15/2024.

The collected data reveal that the scalability wall of the RCN mainly (94%) manifests eight symptoms as listed in Table 1. They occur when an RNIC's aggregated throughput or latency is not at the expected level as in their official specifications (referred to as a *performance issue*). These symptoms come from different RCN layers such as the virtual switch (i.e., Open vSwitch), the kernel driver, and hardware, but are surprisingly all related to RNICs. In total, they have caused 13,396 performance issues during our monitoring period.

- Symptom S1 manifests on the virtual switch when there are 7K+ container flows running over an RNIC. The victim flows repetitively fall back to software processing, thus incurring significant performance degradation (>18×).
- The interactions between the RNIC's driver and hardware can cause kernel stagnation (S2) or crash (S3), especially when the driver creates or deletes flows, thus blocking the execution of all containers on the victim host.
- The remaining five symptoms occur at the layer of RNIC

Table 1: RNIC-related symptoms of the "scalability wall" in our production RCN, as well as their most likely causes.

No.	Symptom	Layer	Ratio	Most Likely Cause (C1 – C8)
S1	Repetitive flow re-offloading	Virtual Switch	17.1%	Flow entries in the RNIC are deleted although they are not aged.
S2	Kernel stagnation	RNIC driver	5.9%	The driver cannot handle the timeout of the RNIC's executing an operation.
S3	Kernel crash on new flows	RNIC driver	5.2%	The driver frees a null pointer when the RNIC fails to create a flow entry.
S4	Slow flow state maintenance	RNIC hardware	11.4%	Flow deletion in the RNIC takes much longer (9×) time than expected.
S5	Intermittent software forwarding	RNIC hardware	15.3%	Flow counters are not updated timely. The virtual switch reads a "dirty" value.
S6	Poor performance of specific flows	RNIC hardware	29.9%	Flow entries with different masks are queried sequentially in the RNIC.
S7	PCIe link down when unbinding VFs	RNIC hardware	8.4%	Race condition emerges when the RNIC cleans up allocated resources.
S8	RNIC unresponsiveness	RNIC hardware	6.8%	VXLAN encapsulation contexts exceed the RNIC's buffer capacity.

hardware. For example (S6), a specific flow's performance can be persistently poor when ∼10K flows are offloaded to the RNIC hardware (we did not discover any anomalies in the software stack). Even if the workload drops off greatly (e.g., from ∼10K to 600 flows) afterwards, the performance cannot recover in a short time.

From Symptoms to Causes. Given the various symptoms, we wish to figure out their root causes, so that we can take measures to proactively optimize the performance or at least prevent further degradation. Unfortunately, we encounter a key challenge that we have very limited visibility into the internals of today's commodity RNICs [32,36]. The internal design of RNICs is mostly close-sourced and hardly mentioned in their specifications. Resorting to the vendors also turns out to be ineffective in most cases, since they have neither encountered nor understood such "elusive" symptoms. After all, these symptoms are only manifested in large-scale deployments that involve extremely complicated workloads/environments and push the RNIC's capability to its limit, which are not easy for the vendors to reproduce and analyze. Furthermore, we are not allowed to share real-world workloads with the vendors, making their troubleshooting difficult, if not impossible.

To address this, a more pragmatic approach is to *infer* the most likely causes according to common abstractions of an RNIC's components and functionalities. An RNIC on a host provides RDMA verb functionalities (e.g., `read` and `write`) to each hosted container, which allow the container to operate on the resources in the RNIC's conceptual components (e.g., send/receive queues). Besides, today's RNICs in data centers all implement *match-action* based embedded switch (or *eSwitch* for short) components to support hardware-accelerated packet switching [38,41,55], which are widely used for efficient container network virtualization. Our approach fundamentally differs from the existing blackbox testing techniques [67] or whitebox analyses [54], as it leverages the domain knowledge of RNICs' common architecture to construct a highly likely RNIC model for actual testing and optimization, and thus is more of a "greybox" approach.

With the common abstractions of an RNIC, however, we are still confronted with a prohibitively enormous search space, stemming from the components' possible combinations and the functionalities' possible configurations. For example, highly configurable eSwitch tables can have $O(k^n)$ possible matchers, where n is the number of tables and k is the average number of table entries. We conduct *combinatorial causal testing* to overcome the difficulty in a principled way. First, we leverage topological restrictions across components to efficiently construct valid component combinations (i.e., the architecture model of the RNIC). This is achieved by inspecting components' dependencies to filter out the combinations that lead to packet loops or unreachability, which reduces the magnitude of combinations to a quadratic polynomial level.

We then test the RNIC's functionalities with real and synthetic workloads on each valid architecture model. Once a symptom occurs, we perform a *local sensitivity analysis* [19] to deduce the causal relations between the RNIC's input (workload) and manifested performance. Specifically, we strategically tune the value of each dimension (e.g., the number of table entries) in the workload to test whether the symptom can be eliminated or alleviated. If so, the component(s) related to this dimension are a likely cause of the symptom, which might lie in a critical path of the RNIC's packet processing pipeline. Next, we fine-check each concerned component in the critical path by means of *permutation removal* [15], which eliminates the component whose removal would not influence the performance. In this way, we sort out the critical path(s), pinpoint the most likely causes, and effectively approximate the RNIC's performance model for a valid architecture model. Eventually, we choose the architecture model that best matches the actual performance as the final reference.

Optimization, Evaluation, and Validation. Based on the above efforts, we are able to proactively optimize the NF offloading schedule for every RNIC, by reorganizing the offloaded flows to minimize the estimated processing time over the critical path for each flow. Also, we avoid triggering the inferred design/implementation defects of the RNIC by transforming the risky functionality inputs to their equivalents that are unlikely or less likely to cause performance issues.

We implement the whole design into a practical system dubbed ScalaCN, and conduct extensive experiments based on real-world production workloads with six different kinds of RNIC devices (NVIDIA ConnectX-4, -5, -6, -7, BlueField-3, and Intel E810). On average, ScalaCN improves the end-to-end bandwidth by 1.4× and reduces the latency by 31%, after resolving 82% of the inferred causes of performance issues (the remaining 18% are due to hardware limits). Its major overhead comes from continuous monitoring and optimization operations, with <5% usage on a single CPU core.

We have reported all performance issues and their most likely causes to relevant vendors, all of which have been encouragingly confirmed. In detail, the vendors have released patches on the driver and firmware to fix the root causes C1, C2, C3, and C5. For the remainder (C4, C6, C7, and C8), we are now closely collaborating with the vendors to fix them as they involve the intricate co-tuning of software and hardware.

Contribution. This work makes the following contributions.

- We conduct the first study to uncover the scalability wall in a large-scale RCN, and pinpoint the culprit to be RNICs.
- We conduct combinatorial causal testing based on RNICs' common abstractions, to efficiently approximate their internals and infer the root causes of performance issues.
- We devise an effective method to accommodate RNICs to RCN scaling. Evaluation on real-world workloads and the feedback from vendors confirm its efficacy. We are now gradually deploying ScalaCN over the production RCN.

The code and data involved in this work have been released in part at https://scala-cn.github.io.

2 Background

Container networks enable transparent data communications among containers, and they are currently realized in three major approaches [35, 70] as listed in Table 2. Different approaches provide different levels of connectivity among containers and different levels of isolation with the host network. Today's mainstream container services widely adopt the overlay (or overlay-like) mode to organize container networks [2, 4, 7, 13, 26], since this mode enables accessibility across hosts while preserving isolation.

Figure 1 shows the typical architecture of an overlay-based container network. Two hosts (A and B) are connected via physical switches/routers in the underlay network. Each overlay container on a host has its own IP address within the same VXLAN [42] subnet. For example, a container on Host A has the IP address 172.16.122.1 within the VXLAN subnet 172.16.122.0/24. When two containers communicate with each other, the software virtual switch on the host OS will determine to which port the packet should be forwarded, and perform VXLAN encapsulation for packets (if needed). For container communications across hosts, the encapsulated packets will go through the hardware NIC and the underlay network.

Hardware Offloading with RDMA NICs. In practice, manipulating and forwarding container packets with software switches is resource-consuming. This is because container applications (e.g., large model training) often generate a large volume of network traffic which is processed by the kernel [3, 6] and the user-space software switch, incurring significant CPU overhead and processing latency [22, 39].

To alleviate the packet processing overhead on the host OS, today's mainstream container service providers (CSPs) take

Table 2: Different modes of container networking, where "isolation" refers to *network* isolation from the host OS.

Mode	Mechanism	Connectivity	Isolation
Bridge	Virtual bridge	Host only	In-bridge
Host	Namespace sharing	Cross-host	No isolation
Overlay	VXLAN	Cross-host	In-overlay

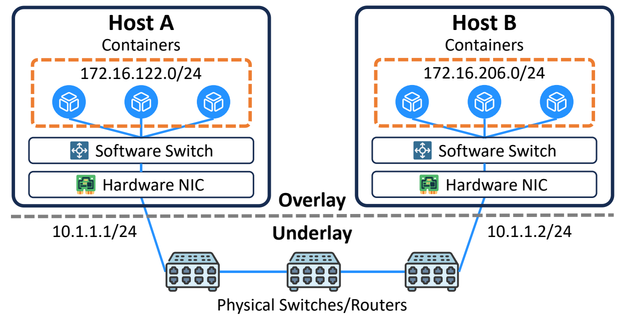


Figure 1: Example of overlay-based container networks.

advantage of the hardware offloading functionality [38, 47] with the single-root IO virtualization (SR-IOV) technique of RDMA NICs (RNICs) [28, 63]. This brings two significant performance benefits. First, all the containers can communicate with each other through the RDMA protocol, without needing the intervention of the OS. Second, the CPU-intensive tasks such as packet forwarding and VXLAN encapsulation also bypass the software stack, i.e., they are all moved from CPUs to RNICs for more efficient processing.

The resulting RDMA-offloaded container network (RCN) significantly improves the network performance. A representative example is illustrated in Figure 2. When the traffic is delivered using TCP, the average end-to-end bandwidth is 21.17 Gbps and the average packet forwarding latency is 87.31 us. When VXLAN encapsulation and packet switching are offloaded to an NVIDIA ConnectX-6 RNIC, the bandwidth grows to 50.31 Gbps, and the latency drops to 54.51 us. If we further use the RDMA protocol to transmit packets, the bandwidth remarkably increases to ~180 Gbps and the latency decreases to merely 2.7 us.

3 Motivation

Despite the merits of RNIC offloading, an RCN's performance seems to be subject to its scale—during our one-year operations, we noticed that once the network workload hits a "scalability wall", a variety of performance issues occurred and severely hurt the QoS (Quality of Service).

3.1 Continuous Monitoring Infrastructure

To deeply understand the scalability wall, we built a continuous monitoring infrastructure on top of our large-scale production RCN, which carries ~1,500 PB of traffic for ~0.5 million containers every day during our monitoring period (from Apr. 15th, 2023 to Apr. 15th, 2024).

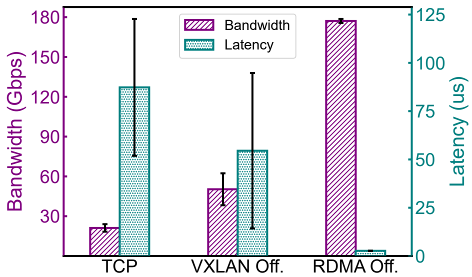


Figure 2: Performance comparison with and without hardware offloading (Off.) in a container network.

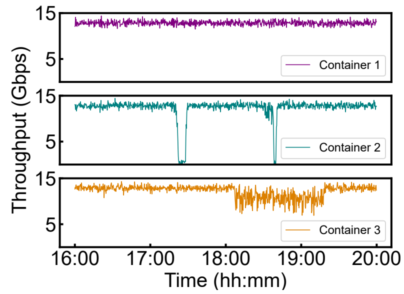


Figure 3: Performance issues caused by S1, where the virtual switch repetitively re-offload flows.

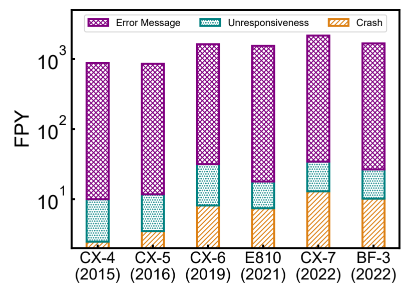


Figure 4: Kernel instability in terms of failures per year (FPYs). Here the RNICs are sorted by release date.

All containers in our RCN are Docker [9] containers. We use Kubernetes [10] to orchestrate Docker containers together with the container network interface (CNI) [8] plugin that we developed in-house. Each host in our RCN is equipped with one to eight RNICs, which are NVIDIA ConnectX-4 (CX-4), ConnectX-5 (CX-5), ConnectX-6 (CX-6), ConnectX-7 (CX-7), BlueField-3 (BF-3), or Intel E810 (E810). We balance the load of these RNICs when binding them to containers. We deploy Open vSwitch (OVS) [46] on each host as the software virtual switch to handle packet forwarding. When containers initiate a flow, the OVS will use the `TC` [5] utility to offload packet processing tasks to the RNIC.

Our continuous monitoring infrastructure gathers basic performance data across different layers on each host as follows, so that we can be aware of the RCN status in time. These data are transformed into concrete states of network components, and indexed in our log services for aggregation and analysis.

- *RNIC and Kernel Statistics.* We collect packet sending and receiving rates as well as packet drops of the RNIC through Linux `sysfs`. We also collect logical flow entries recorded by the kernel using `netfilter` and `conntrack` to track the states of all RDMA connections.
- *OVS Offloading Status.* The OVS determines what packet manipulations should be applied to an RCN flow and whether it needs to be offloaded to the RNIC. To monitor these events, we gather OVS information including table hits, misses, and losses of flow lookups [46].
- *CNI Events.* We collect crucial events on resource allocations triggered by the CNI plugin. We monitor the creation of containers, as well as the allocations of VXLAN subnets, IP addresses, and VFs (virtual functions that containers are one-to-one bound to) of RNICs.

3.2 Symptoms of Performance Issues

Our collected data reveal 14,251 performance issues regarding the scalability wall of the production RCN during our monitoring period. The eight major symptoms are listed in Table 1, accounting for 94% of the performance issues. Although these symptoms come from different layers of the RCN, such as the virtual switch, the kernel driver, and the hardware, they are all related to the RNICs which were supposed to bring high-performance data communications among containers.

Virtual Switch. When the number of flows offloaded to an RNIC increases to a threshold, e.g., 7K+ over a CX-6 RNIC, the virtual switch would repetitively re-offload the flows (Symptom S1). This accounts for 17.1% of RNIC-related performance issues. When this occurs, the end-to-end performance on the affected containers greatly fluctuates.

Figure 3 shows our production traces regarding S1. The three containers are used for training the same AI model with the same configurations (e.g., QoS and VF quotas). The throughput of Container 1 keeps stable at around 15 Gbps, which is the expected performance. However, Container 2 and Container 3 both suffer from performance degradation and present two different patterns of performance issues. The throughput of Container 2 periodically drops to <1 Gbps (which is nearly $18\times$ worse) and then recovers to $\sim$15 Gbps. The throughput of Container 3 becomes unstable, sometimes fluctuating between 5 Gbps and 10 Gbps for an hour. These issues significantly affect the training process of AI models and oftentimes lead to users' complaints.

By analyzing the logs of the OVS, we find that the OVS repetitively re-offloads the flows for Containers 2 and 3 due to constant lookup misses in the RNIC's flow tables, as if the RNIC is "refusing" to handle them under the high workload.

RNIC Driver. We also find that in our production RCN, the RNIC's driver often makes the kernel unstable when it interacts with the RNIC's hardware, which manifests as S2 and S3 in Table 1. When handling a large number of flows offloaded in a short time (e.g., 1K per second), the kernel can become stagnated (S2) after the driver sends certain operation codes to the RNIC. Through detailed timing on the driver's call stack in the kernel, we observe that the driver seems to be always waiting for the RNIC's response, which however cannot be received in time under the high workload. Worse still, the kernel can even crash (S3) when the driver fails to offload the flows to the RNIC. Even though these symptoms occur less frequently (11.1% in total) than the symptom oc-

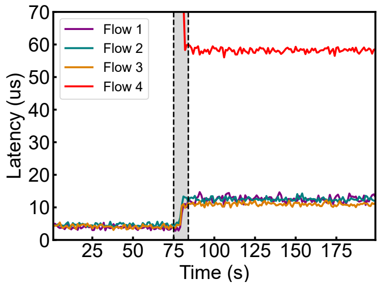
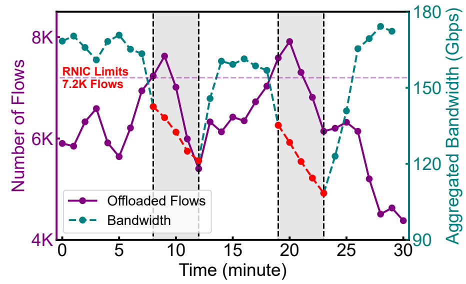
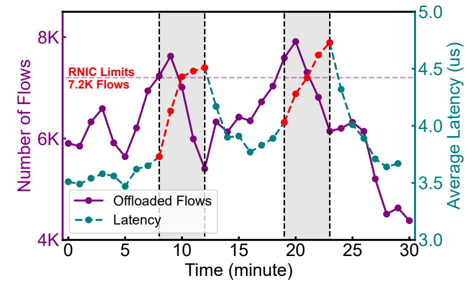


Figure 5: Latency increases when a specific flow (Flow 4) is offloaded.



(a) Aggregated bandwidth.



(b) Average packet forwarding latency.

Figure 6: Workload impacts on the RDMA `write` performance of the CX-6 RNIC.

curring at the virtual switch layer, they are more severe since they make all the containers over the host unavailable.

Figure 4 shows the kernel's failures per year (FPY) with regard to different (models of) RNICs in production. RNICs released recently (e.g., E810, CX-7, and BF-3) bring much more FPYs than those released eight years ago (e.g., CX-4 and CX-5) – note that the Y-axis uses a log scale. The reasons are mainly two folds: 1) newer RNICs have more data to be synchronized, and 2) newer RNICs' kernel drivers are more complex but not time-tested enough. In fact, undesired symptoms regarding the RNIC's driver have not been discovered by the vendors as they did not (and have no means to) test the driver and RNICs under a large-scale production RCN.

RNIC Hardware. The majority (71.8%) of RNIC-related performance issues occur at the hardware layer, which are mainly due to the RNIC's internal defects as we did not find any anomalies in the software stack (i.e., the virtual switch and the driver) when these issues occur. Their root causes are much harder to locate and identify, since today's commodity RNICs are usually a black box to their users.

When network flows are offloaded to an RNIC, the software stack needs to maintain the flow states (e.g., to invalidate the flows when they are aged). During this process, we find that state synchronizations from the control plane to the RNIC's hardware can become extremely slow, making the software unable to offload new flows in time and therefore causing performance degradation (S4). Further, even if the control plane has offloaded the flows to the RNIC, these flows can intermittently go into the software stack for being processed, indicating that the RNIC is not functioning properly (S5).

Moreover, when the workload on the RNIC reaches a certain threshold, the performance of specific flows can become extremely poor, which would in turn affect the performance of other flows (S6). As shown in Figure 5, before the offloading of Flow 4, Flows 1–3 all have a desired latency of $\sim$4 us on the CX-7 RNIC. When the new Flow 4 is offloaded at around 75s, it does not reach the desired packet forwarding latency of $\sim$4 us but instead bears a high latency of $\sim$60 us. Meanwhile, the latencies of Flows 1–3 also increase by 3$\times$.

Another example of S6 is shown in Figure 6. It depicts the performance of the RDMA `write` verb of the CX-6 RNIC with different levels of workloads on it. When there are fewer than 7,200 flows offloaded to the RNIC, the performance is in general negatively correlated with the workload. In contrast, when over 7,200 flows are offloaded, as shown in the shadowed areas of Figure 6a and Figure 6b, the RNIC's aggregated bandwidth instantly drops by 13%–37% and the average latency increases by 27%–34%. Even when the workload drops off to a very low level (e.g., only 600 flows), the performance of the RNIC can hardly recover in quite a while (e.g., 6 minutes) but continues to degrade (cf. §4.2).

For the last two symptoms at the hardware layer (S7 and S8 in Table 1), they are more severe than S5 and S6 since they can make all the containers over the host unavailable. For example, when we de-allocate the VFs of an RNIC from the containers, we observe that the PCIe link state sometimes goes down. Also, when we offload too many VXLAN encapsulations to the RNIC (e.g., 10K on the BF-3 RNIC), the RNIC can become unresponsive and unable to handle any flows. Since our RCN supports multi-tenancy, the number of flows can easily reach 10K+ on a single RNIC, which makes this issue very common in production.

3.3 Challenges

According to the long-term monitoring experiences, we notice quite a few performance issues regarding the scalability of the RNICs in production, where RNICs present undesired performance compared with their official specifications. Thus, CSPs desire an effective approach to understanding the practical limitations of RNICs to prevent performance issues in large-scale deployments. Unfortunately, we have very limited visibility into today's RNICs as their internal design and implementations are usually non-public. Although some of the existing studies have conducted comprehensive tests on commodity RNICs [32, 36, 37, 67], they can only discover performance issues but can hardly tell the root causes and the solutions. Besides, it is rather difficult for RNIC vendors to reproduce and understand such "elusive" symptoms happening in large-scale data centers that involve extremely complicated workloads and environments. In addition, we are not allowed to share real-world workloads with the vendors, making their troubleshooting very difficult.

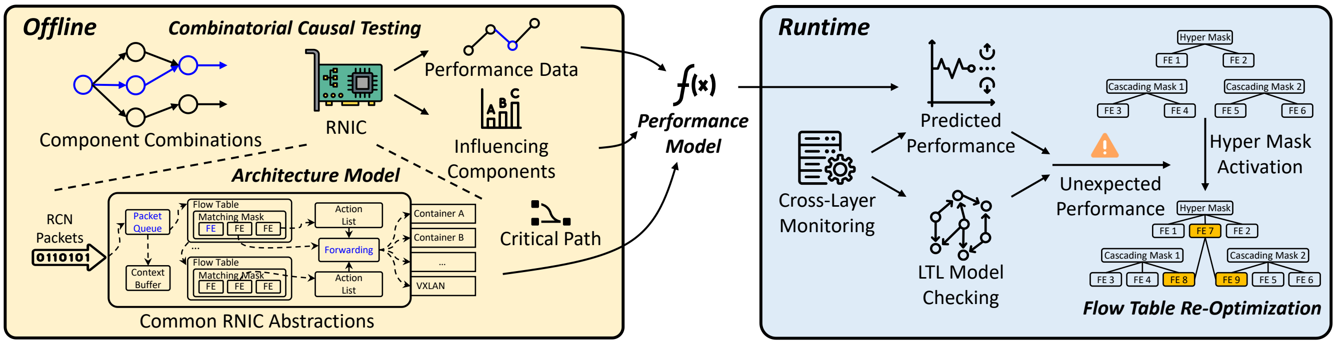


Figure 7: Architectural overview of ScalaCN, a scalable RCN based on the efficient approach of combinatorial causal testing. The blue color of the entities in the offline phase represents their causal relations as a typical example.

4 Design and Implementation

We present ScalaCN, an effective and efficient system for understanding and optimizing RNIC performance in a large-scale RCN through a greybox-like approach. Our key insight obtained from §3 is that the widely existing performance issues on the RNIC's scalability originate from the design and implementation defects of RNICs, which highlights the importance of dissecting the RNICs' internals when these issues occur. Due to the usually non-public implementations of the RNICs, we strive for a more pragmatic approach by *inferring* the most likely causes of the issues according to the common abstractions of an RNIC's components and functionalities.

Figure 7 shows the architecture of ScalaCN, which consists of the offline phase where the combinatorial causal testing is conducted on the RNIC, as well as the runtime phase for RNIC performance prediction and optimization.

- *Combinatorial Causal Testing* (§4.1). In the offline phase, our goal is to reason about the architecture model and approximate the performance model of RNICs based on their common abstractions. However, we are confronted with a prohibitively enormous search space of an RNIC's possible component and functionality combinations. To address this, we first use the topological restrictions among the components to filter out invalid combinations. This significantly reduces the complexity of building architecture models for an RNIC. With respect to each valid architecture model, we further approximate the RNIC's performance model by inferring probable critical paths of the RNIC's packet processing through local sensitivity analysis [19] and permutation removal [15]. Eventually, the most likely performance model becomes our final choice.
- *Performance Interpretation and Prediction* (§4.2). Based on the approximated RNIC's performance model, ScalaCN collects the runtime data of the RNICs and makes the actual predictions on the performance. In this way, ScalaCN prepares to proactively schedule the offloaded network functions. The guidance of the performance model inferred by the combinatorial causal testing enables us to achieve a high prediction accuracy of 98%. We further repair the 2% false predictions by temporal-logic model checking [18, 49].
- *On-Demand Performance Optimization* (§4.3). We do not need the RNIC to always work in its perfect status, but once it is anticipated to undergo a noticeable (empirically set as 5%) performance degradation, we optimize the network function offloading by reorganizing its flow rules to minimize the possible critical path of packet processing in the RNIC hardware. This is achieved by reordering flows with a strong locality into the headmost matching masks in the flow tables that have the lowest delay.

4.1 Combinatorial Causal Testing

Common Abstractions of RNIC Components. As we have mentioned in §2, an RNIC in a production RCN accelerates data communications among containers mainly from two aspects, i.e., RDMA support and packet switching offloading. Specifically, an RNIC provides RDMA verbs (such as the `send`/`receive` and `read`/`write` primitives) to enable each hosted container to use the RDMA protocol to transmit packets without the need for the intervention of the OS. This is typically achieved with the SR-IOV technique [28, 63], where the RNIC is virtualized into multiple virtual functions (VFs) and each VF can be bound to a container.

Besides, an RNIC accelerates the packet switching tasks involved in the RCN. Recall that in Figure 1, an RCN host must be equipped with a virtual switch to decide to which containers a packet should be forwarded (i.e., the switching rules). Also, the virtual switch needs to encapsulate the packet with the VXLAN header (i.e., the applied action) to ensure the packet can cross the underlay network if necessary. These tasks could incur significant overhead if they are all processed in the software stack. Thus, they are offloaded to the RNIC's hardware in the RCN through `TC` or `DPDK` utilities [48].

In other words, the RNIC acts as a programmable switch with RDMA supports to forward packets among the overlay and the underlay. In fact, today's RNICs in data centers all implement *match-action* based embedded switch (or *eSwitch*

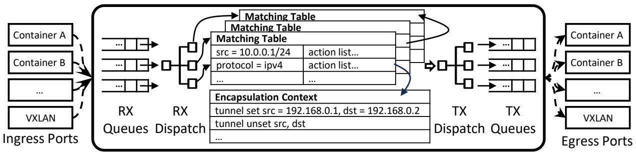


Figure 8: An RNIC's common (abstracted) components in the RCN. The packets come from containers or VXLAN, which are processed by the RNIC and sent to destination ports.

for short) components to support hardware-accelerated packet switching [38, 41, 55]. This has become the de-facto standard industrial practice and conforms with the kernel's *switchdev* model [1] for handling the data plane in the hardware while keeping the control plane unmodified [12].

Given the above functionalities, we abstract the RNIC's components as shown in Figure 8. Note that we only focus on the *data path* of the RNIC (i.e., the pipeline that processes the packets) since it is the most related to the packet processing performance. Specifically, each container binds to one or more VFs of the RNIC, with which the container can use the RDMA verbs to request the RNIC resources (e.g., queue pairs) and transmit packets through high-performance RDMA protocols. When a packet from the container is received by the RNIC through the VF, it will be enqueued into the container's requested queue pairs and then dispatched to the RNIC's eSwitch component. The eSwitch component is composed of a number of flow tables to match the packets and then apply the recorded actions on them. These tables are offloaded from the control plane in the software stack.

Search Space on Component Combinations. After abstracting the RNIC's components, we aim to infer the likely architecture models of the RNIC. In particular, we need to search for the combinations of components/configurations that are valid inside the RNIC under the environment of our production RCN. If we have such information, we can further reason about the more detailed performance impacts of the RNIC's (abstracted) components and configurations.

Unfortunately, we are faced with a combinatorial explosion of the possible combinations of the RNIC's components. Today's RNICs are highly programmable and can be offloaded with a large number of combinations of flow tables and RDMA flows. Even for a single flow table offloading, the RNIC can have hundreds of flow entries, each of which can have multiple matching rules and actions. For example, the matcher of a flow entry for NVIDIA CX series RNICs is at least 192 bits long [12], which can cover different protocols, sources, and destinations, from the MAC layer to the transport layer. Each bit of the matching mask is individually configurable. Further, flow tables can be chained together through the forwarding action in each entry of the table for a more flexible packet processing pipeline in the data path, resulting in a more complex structure of the RNIC's components and their interactions during the RCN packet processing.

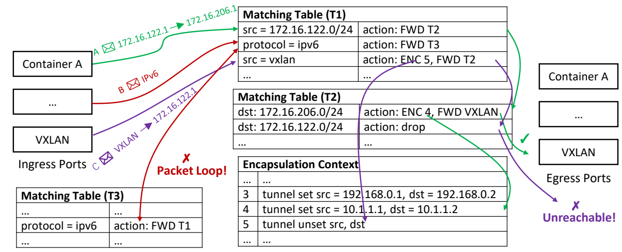


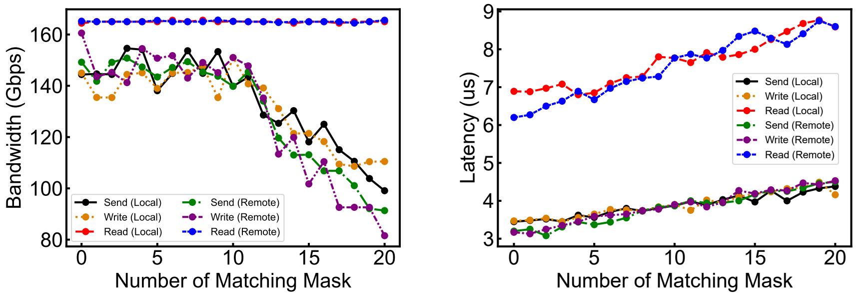
Figure 9: Examples of topological restrictions on the combinations of components.

When putting all possible combinations together, the search space of the combinations is prohibitively large, i.e., $O(k^n)$ possible matchers, where n is the number of tables and k is the average number of table entries. Note that in a production RCN, the actual value of n and k can all be as large as 100.

Efficient Searching with Topological Restrictions. In order to reduce the probable combinations of the abstracted RNIC's components and their configurations, we leverage the *topological restrictions* among the components to filter out invalid combinations. Our key idea is that the RNIC's components and configurations are not independent but have strong dependencies and restrictions among them. These restrictions are brought by the network topology and packet reachability.

As shown in the examples of Figure 9, not all the combinations of rules in the table can correctly deliver the packets to the desired destinations. The only valid combination in the figure is the entries of flow tables that packet A goes through. These flow rule configurations enable the RNIC to deliver the packet from container A to the VXLAN interface (i.e., the underlay network). In contrast, the table entries related to Packets B and C violate the topological restrictions. For Packet B, the example combination on flow entries leads to packet loops between flow tables T1 and T3. This is because the IPv6 packets will be forwarded to flow table T3 when they are matched by flow table T1, while T3 will then forward the packet back to T1. Thus, such a combination of flow table entries is not valid in practice. Similarly, Packet C will be forwarded to an unreachable destination, and as a result the corresponding combinations are also invalid. Eventually, we can simply test the eSwitch and queue pair combinations that enable correct end-to-end packet delivery among the containers (i.e., the architecture models of the RNICs), thus reducing the magnitude of component combinations and configurations to $O(k \cdot n^2)$, where k is the number of subnets and n is the average number containers in each subnet.

After pinpointing the valid architecture models in our production RCN for the RNIC, ScalaCN generates test traffic to search for relevant symptoms as we have encountered in Table 1 on each architecture model. To this end, ScalaCN starts to test the flow table combinations from a simple case, i.e., only one flow table. Then, it continuously adds new flow entries to the table with different matching masks, IPs, ports,



(a) Aggregated bandwidth.

(b) Packet forwarding latency.

Figure 10: Sensitivity analysis on the number of matching masks for CX-6.

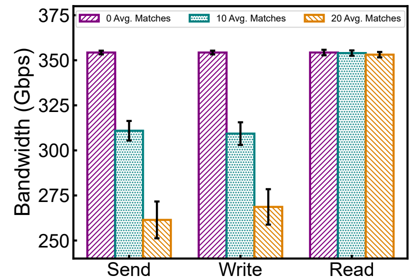


Figure 11: Impacts of the avg. flow table queries of concurrent in-flight packets.

protocols that are reachable in our production RCN, so as to test the RNIC's performance. It also adds forwarding actions to the packet processing rules to connect two flow tables. These rules are installed to the RNIC through the `TC` [5] utility. All these manipulations are conducted with a guarantee that the packets can be correctly delivered. Since we have already filtered out the invalid combinations of the RNIC's components, such testing can be done efficiently and effectively.

Causal Inference. Once the above searching process finds the symptoms (i.e., the performance issues) to occur, ScalaCN conducts a *local sensitivity analysis* to pinpoint the causal relations between the components and the performance issues. ScalaCN incrementally changes each dimension of the input configurations and tests the performance fluctuations. These dimensions come from the test configurations as we mentioned above. During this process, ScalaCN uses the spring-based control model [66] for congestion avoidance to achieve efficient bandwidth testing. If the performance issues can be alleviated or exacerbated after changing the configurations (e.g., increasing or reducing flow tables), we can infer that the corresponding components related to this dimension are likely to lie in the critical path of the RNIC's packet processing.

Here we show an example of the use of local sensitivity analysis to find the possible components that are located in the critical path of the RNIC's packet processing. At first, the searching process of ScalaCN finds that the CX-6 RNIC's performance degrades for certain flows when the number of flow tables increases, whose symptom is similar to S6 in Table 1. However, it is not easy to infer which configurations are likely to be the root cause of the performance issue.

ScalaCN then incrementally changes each testing dimension and examines the resulting performance. Figure 10 shows the performance of CX-6 when a single dimension (i.e., the number of matching masks in offloaded flow tables) of the workload is changed. ScalaCN quickly finds that the RDMA `write` and `send` bandwidth of the CX-6 RNIC is significantly affected by the number of the matching masks in the flow tables, while the `read` verb is hardly affected.

Specifically, the match-action table in RNICs matches packets based on the packet patterns and determines how to process the packet. The offloaded table entries consist of two parts, i.e., the matching mask[1] and (multiple) matching values. The mask indicates which fields of the packets should be matched, while the values specify the content that the packet fields should be if matched. Flow entries with the same mask are grouped and queried together. When the number of masks offloaded to the RNIC increases, ScalaCN finds new flows matched by the newly offloaded masks (i.e., the mask has not been offloaded before) will have a lower performance in the packet processing. Thus, we can infer that the matching masks in the flow tables are likely to lie in the critical path in the RNIC's packet processing. Nevertheless, till now we are still not able to pinpoint the exact components or configurations that really form the critical path in the RNIC.

In order to refine the probable components in the inferred critical path of the RNIC, ScalaCN further performs *permutation removal*. The key idea of this step is to eliminate the components lying in the possible critical path whose removal does not affect the performance of the RNIC. It first sets up a hypothesis H_0 that assumes a specific dimension d is significant to the performance. Then, it permutates the concrete values of the other dimensions, and records the corresponding RNIC's performance. If the probability of the performance degradation is $>95\%$ compared with the tests when d is not set to the current value, we determine that the dimension d is significant (i.e., accept the hypothesis H_0), otherwise we remove the dimension d (i.e., reject the hypothesis H_0)

In this way, we make the critical path more concrete and sort out the real critical paths of the RNIC's packet processing, which guide us to reason about the performance issues and optimize the RNIC's performance. We choose the architecture model with the best accuracy as the final reference.

4.2 Performance Interpretation & Prediction

In this section, we provide detailed interpretations for the symptoms in Table 1, which are then used to predict the

[1]Different vendors have different terminologies on flow tables. NVIDIA names the matching mask as the flow group, while Intel names it as the matching recipe. Although we use the term "matching mask" throughout the paper, its actual meaning is the same as flow group and matching recipe.

RNIC's performance. We conduct a sequential analysis [60] on the packet processing pipeline of the RNICs to validate how packets are delivered and coordinated inside the RNICs.

Flow State Maintenance. We find the possible causes of the state inconsistency problem in the RNICs. First, we note that the flow deletion in the RNICs can sometimes take a much longer time than expected. This symptom is caused by the flow table organization – when a flow is deleted, the RNIC must search the flow table to find the possible flow table dependencies on the specific flow; as a result, if the flow table is large, the deletion process becomes slow, causing S4. Similarly, the flow counters are not updated in time when the RNIC is handling a large number of flows. This makes the software stack determine that the corresponding flow is inactive. Thus, it may delete the offloaded flow from both the software and hardware by mistake, which leads to S1 and S5. These issues mostly happen to the NVIDIA devices.

Flow Tables. With the understanding of matching masks and flow table queries, we demystify the RNIC's performance in Figure 10. For the `read` verb, the local host first requests the remote host's memory address [65]. Its latency increases when the number of local hosts' egress (or remote's ingress) matching masks increase since requesting the addresses involves sending request packets from the local to the remote, which *sequentially* examines the matching masks for the packet's local encapsulation and remote decapsulation.

After the remote host's addresses and keys are requested, the `read` verb only involves packet transmission from the remote to the local [65]. As the number of the remote host's egress-direction masks (or the local host's ingress-direction masks) does not increase, the `read` bandwidth remains stable at 165 Gbps. In contrast, for `send` and `write` verbs, all packets are from the local to the remote. Their bandwidth and latency all degrade when egress-direction masks at the local or ingress-direction masks at the remote increase.

The above also explains the S6 in Table 1. When a matching mask is created, the RNIC sequentially queries the flow table entries. A new mask is queried after the old one, which incurs additional mask query latency. Even if the workload drops off later, the new flow still belongs to the new mask (i.e., it cannot be moved back to the old one which has a lower delay) and the performance cannot recover. We have observe such issues in all the examined NVIDIA and Intel devices. Besides, although the flow tables and matching masks are examined sequentially in the RNIC, the performance of flow matching under the same mask is nearly the same. This indicates that matching values in the RNIC are organized by hash maps under the same group partitioned by the matching mask.

The above micro behaviors of RNICs are actually invisible to the software stack (e.g., OVS) without ScalaCN. As a result, it is difficult for the software stack to schedule the optimal offloading of flow tables and matching masks, which often leads to inferior networking performance in the RCN.

Packet Queuing. We note that before a packet queries flow tables inside the RNIC, it will go through the requested queue in the RNIC. Packet queuing limits concurrent packets processed by an RNIC at the same time. When there is only one flow passing the RNIC, all packets of this flow fill the queue resources and fully utilize the RNIC capacity. However, when there are many flows, queue resources are shared by these flows' requested queue pairs. The bandwidth of a flow is proportional to its allocated queue pairs.

The queued packets affects the RNIC's overall performance jointly with matching mask queries. Figure 11 shows the impacts of queued packets of the CX-7 RNIC. When the average matching mask queries of queued packets increase, the `send` and `write` performance degrades, while that of the `read` verb is still hardly affected as CX-6. For latency, all three verbs are affected by the average matching mask queries similar to the one-flow case (not shown). In other words, an RNIC's overall performance is influenced by the average matching mask queries of queued in-flight packets.

Action Lists and Resource Management. When a packet is matched to a flow table entry, the RNIC applies actions (e.g., encapsulation and forwarding) recorded in the entry. Action lists incur additional delay on packet processing. We notice that VXLAN encapsulation and decapsulation incur performance loss (e.g., $\sim$6% on throughput and $\sim$3% on latency of all examined RNICs). More actions applied on the packet lead to a lower performance. However, as all packets in the overlay-based RCN only require VXLAN actions and port forwarding, such performance loss is constant in our scenario. Further, when the number of offloaded VXLAN actions increases, the corresponding buffers in the RNIC can be exhausted, leading to RNIC unresponsiveness (S2 and S8).

We have reported all the above findings and the inferences on the RNIC's hardware to the relevant vendors, which have been all confirmed. In particular, the inferred causes of S1, S2, S3, and S5 have all been fixed with the driver and firmware updates from the vendors. For the remaining causes, we are closely collaborating with the vendors to fix them.

Performance Prediction. The above combinatorial causal testing provides explainable models for us to understand where the performance issues of RNICs come from. ScalaCN employs the results as a proactive performance predictor to forecast performance degradation. ScalaCN predicts RCN performance on an RNIC basis, i.e., we only focus on monitoring the overall performance of an RNIC.

We use statistical fitting to enable performance prediction on RNICs at runtime. After testing diverse fitting functions including linear functions, polynomial functions, exponential functions, radial basis function [21], and so on, we find that the expected bandwidth *BW* of various RNIC models can be best fitted with a radial basis function as follows

$$BW(Q_l, Q_r) = u \cdot e^{\frac{-(Q_l - m)^2 + (Q_r - n)^2}{2 \cdot v^2}} + w, \quad (1)$$

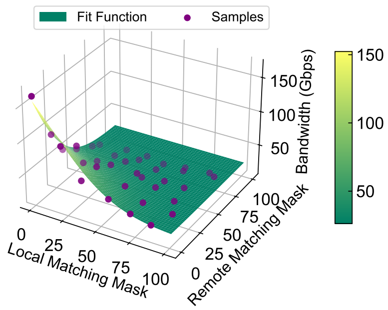


Figure 12: Fitting bandwidth performance of CX-6 with matching masks.

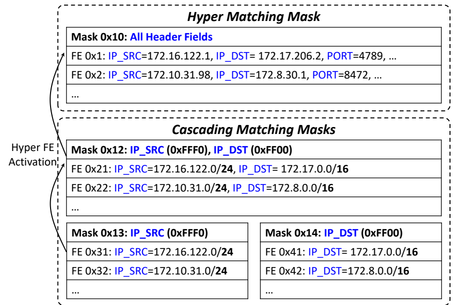


Figure 13: Optimizing RNIC performance by activating the flow entries (FEs).

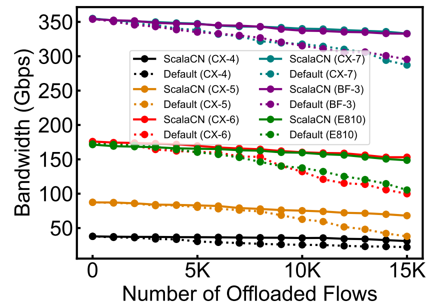


Figure 14: RNIC aggregated bandwidth with and without ScalaCN.

Table 3: Fitting parameters for different RNIC models, where **GD** represents goodness of fit.

RNIC	u	v	m	n	w	a	b	c	GD
CX-4	78.86	39.14	-28.25	-43.19	4.91	0.049	0.063	5.02	0.94
CX-5	148.97	42.34	-29.37	-45.14	12.43	0.051	0.074	4.93	0.93
CX-6	324.47	42.13	-26.92	-49.71	26.28	0.047	0.068	2.69	0.93
CX-7	739.52	48.66	-33.53	-52.88	29.32	0.036	0.045	2.57	0.94
BF-3	748.52	48.01	-33.40	-52.42	30.65	0.037	0.043	2.56	0.94
E810	335.64	43.54	-27.01	-49.55	25.16	0.042	0.069	2.74	0.92

where Q_l and Q_r are average matching mask queries of in-flight packets on local and remote hosts' packet queues, e is the natural base, u, v, m, n, and w are fitting parameters. In particular, u and w control the maximum and minimum bandwidth that an RNIC can reach, and m, n, and v jointly determine the sensitivity of bandwidth changes towards the matching mask queries in the RNIC. Although other packet processing processes in RNICs such as access control also add to performance impacts, they are less dynamic and less significant according to our testing, and thus are also considered as a constant influencing factor w.

The above fitting function conforms with our observation that the bandwidth of RNICs is usually non-linear with a high degree of *rotational symmetry* [24] among the input variables as shown in Figure 10a, where the numbers of matching masks on the local and remote hosts have similar performance impacts. Figure 12 shows the fitting process of CX-6 bandwidth with a high goodness of 0.93.

For the latency of studied RNICs, it can be predicted with a linear function (cf. Figure 10b)

$$LAT(Q_l, Q_r) = a \cdot Q_l + b \cdot Q_r + c, \tag{2}$$

where a, b, and c are fitting parameters. Similarly, a and b jointly determine the sensitivity of latency change towards the matching mask queries, and c implies the constant performance influence of action lists. In fact, a linear function is sufficient for latency prediction as it is the simplest function that can achieve a >90% accuracy. Table 3 shows the fitting parameters of our studied RNIC models.

The average flow table query Q of in-flight packets at time t is calculated as $Q(t) = \frac{1}{n}\sum_{i=1}^{n} q_i^t$, where n is the total number of queued packets, and q_i^t is the number of flow table queries that need to be performed on packet i by the RNIC at time t. The number of in-flight packets is measured by the virtual switch statistics. ScalaCN uses a control-theoretic method, i.e., MPC [45], to keep its monitoring overhead under an acceptable threshold, while quickly responding to RCN events.

4.3 On-Demand Performance Optimization

With the performance predictor, ScalaCN monitors and forecasts RCN performance and states. Recall that the results of combinatorial causal testing indicates the matching masks on flow table queries are the key bottleneck of the performance in all the studied RNICs (cf. §4.1 and §4.2). Thus, once ScalaCN predicts that the performance is about to decline, it optimizes the network function offloading schedule by reorganizing the flow tables so as to reduce the flow table queries.

An RCN can have many packet header patterns for masked matching (cf. §4.1). In our production RCN, a host could have different lengths of IPv4 subnet masks. In practice, the OVS and the kernel offload the corresponding matching mask simply in one shot—once there is a new flow pattern, they will create a matching mask in the RNIC without coordinating with the previous ones (as they are not aware of the RNIC's internals). Thus, the RNIC increases matching masks and degrades performance when flow patterns increase.

Hyper/Cascading Masks. ScalaCN optimizes RNIC performance by minimizing matching mask queries on RNIC packet switching, so as to accommodate RCN scale changes. As shown in Figure 13, ScalaCN reorganizes the RNIC's matching masks in an offloaded flow table into two types, i.e., a *hyper* matching mask and *cascading* matching masks. These two types of masks act like a multi-level cache in a CPU. Each flow table has only one hyper mask ahead of all cascading masks, so that it is queried first with minimum delay.

When there is a new flow pattern offloaded from the OVS and the performance is anticipated to noticeably degrade by an empirical threshold of 5%, ScalaCN creates a corresponding cascading matching mask for it. For example in Figure 13, Mask 0x13 and 0x14 match two new flow patterns (i.e., /24 and /16 IP segments). Once ScalaCN detects a packet between two specific containers matched by existing cascading matching masks, it will *activate* the concrete flow into the hyper matching mask (e.g., FE 0x1 in Mask 0x10). The hyper matching mask performs an exact match on packet headers and is faster than cascading matching masks since it is located at the headmost. In this way, flow packets always only need to query the hyper matching mask after flow creation.

The first several packets of newly created flows can go through cascading matching masks. To further minimize packet queries on cascading matching masks, we re-prioritize them based on two metrics, i.e., the length of its matching masks (LM) and the number of packets in the past 60 seconds it matched (PM). A longer matching mask (which matches packets with a more specific pattern and a narrower range) and a larger number of matched packets indicate that the cascading matching mask handles many packets with a strong locality. We thus use the locality score $LS = PM \cdot LM$ to quantify the priority of cascading matching masks. We move the cascading mask with a higher LS to the front so that an average packet can query the least number of entries. Besides, we warm up the RNIC's flow tables (i.e., offload probable flows) to accelerate flow creation in case of bursty traffic. This is achieved based on a Gaussian mixture model [52] deriving from our long-term analysis of production traffic patterns.

A hyper matching mask can be filled up by flow entries more quickly since each entry can only exactly match one flow. Thus, ScalaCN needs to deactivate aged flow entries from the hyper mask to preserve the on-chip memory for new flows. Here we utilize the least recently used (LRU) policy to update the flow entries in the hyper matching mask. Similarly, we remove a cascading mask in the flow table when it has no active packets for ten minutes. According to our measurements, removing aged entries in the two types of the matching masks will not affect the performance of the RNICs. This is understandable because most packets will hit the hyper matching mask or higher-level cascading matching masks first when lower-level masks are removed.

4.4 Generalizability

Other RNIC Components. We also observe the performance degradation incurred by the resource contention of on-chip SRAM like queue pair context (QPC), which has been reported in previous studies [62, 67]. ScalaCN can also be used to locate the contention of the on-chip memory. To achieve this, we only need to add a configurable "capacity" dimension on the original queue abstractions before modeling the RNIC's components. Our combinatorial causal testing can then be used on these new dimensions by differentiating the tested performance under different workloads. Nevertheless, as a CSP, we can only optimize the RNIC's performance within the scope of the configurable components of the RNICs, while the on-chip memory contention is usually the inherent hardware limits that require re-architecting the RNICs.

Reusability. ScalaCN is designed to be reusable as it is based on the common abstractions of the RNIC's components involved in an RCN. Specifically, all RNICs provide the support for common RDMA verbs and eSwitch, which can be manipulated through the same set of APIs. Also, ScalaCN builds the performance model for the tested RNIC to validate the modeling accuracy in a closed-loop manner, and each fitting parameter in the model can be individually tuned based on the tested performance of the RNIC. The only manual operation is determining the common abstractions of the RNICs in an RCN (which is a one-shot effort), while the rest of the processes can all be automated through ScalaCN's testing framework. With the above principled methodologies, ScalaCN can be easily adapted to new RNICs.

4.5 Implementation

We implement ScalaCN in 38K lines of C/C++ code and 13K lines of Python code. In order to monitor RCN states, we make use of the OVS command line utilities including `dpctl`, `appctl`, and `ofctl` to acquire data path information. In addition, to transform LTL policies into an automaton, we use the LamaConv tool [11] to generate Moore state machines [16]. ScalaCN does not create new RCN flow rules, but works between the OVS and the RNIC kernel driver to determine the equivalent flow table offloading strategies that maximize the RNIC performance. We modify the OVS module for replacing the offloaded rules on the fly.

5 Evaluation

ScalaCN is gradually used in production, so it is evaluated using both microbenchmarks and production workloads.

5.1 Experiment Setup

The basic settings of the experiments are similar to those in §3.1. For microbenchmarks, we use a middle-scale RCN with 50 hosts, each of which is equipped with four RNICs including either CX-4, CX-5, CX-6, CX-7, BF-3, or E810. We directly use real-world traffic generated by the hosted applications (e.g., large model training and microservices) for evaluation. These applications initiate $O(4\text{M})$ different flows in a day on average and carry out 150–400 Gbps aggregated throughput on each RNIC. We measure the packet forwarding bandwidth and latency of RNICs using the RDMA `perftest` utility [14]. Note that we only measure performance among the same model of RNICs since the inter-operation across different models could result in instability or RNIC failures [67]. In fact, the inter-operations of different models of RNICs are avoided in our production RCN.

5.2 Microbenchmarks

Scalability. Figure 14 and Figure 15 show the scalability in terms of the RNIC's aggregated bandwidth and packet forwarding latency. When the number of flows on the RNIC

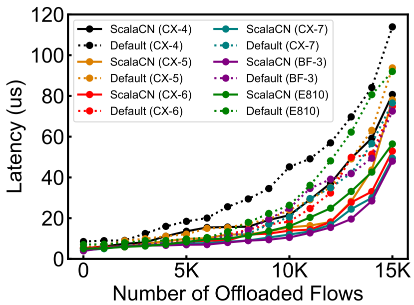


Figure 15: RNIC latency with and without ScalaCN.

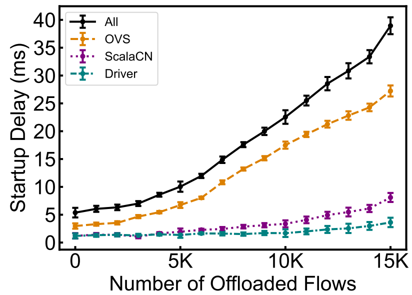


Figure 16: Startup delay of different software-stack components.

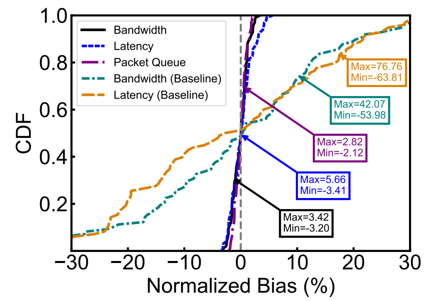


Figure 17: Prediction bias on RNICs' performance.

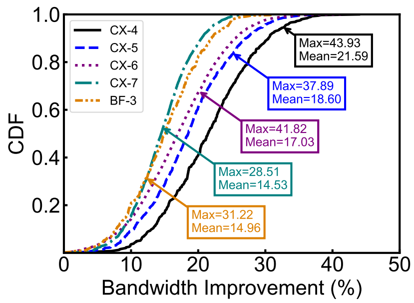


Figure 18: Bandwidth improvement on RNICs in real-world workloads.

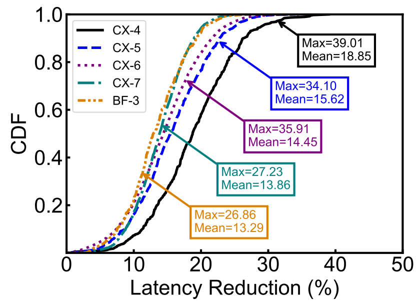


Figure 19: Latency reduction on RNICs in real-world workloads.

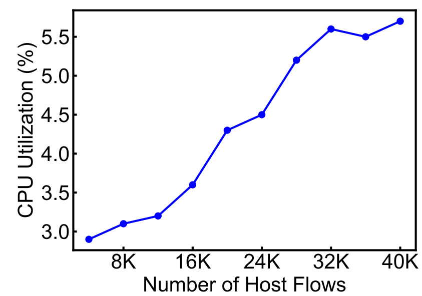


Figure 20: CPU utilization under real-world workloads.

increases, the send bandwidth significantly drops on all models with the default OVS offloading strategy. For example, the aggregated bandwidth of the CX-4 RNIC drops by 41%, and the absolute bandwidth drop of the E810 RNIC reaches 70 Gbps. This is because offloading flows to the RNIC intensifies the resource contention inside the packet queue. Meanwhile, the increasing offloaded flows lead to many matching masks created in the RNIC, which increases the processing delay of a packet and thus degrades the throughput. Further, all RNICs present a threshold after which the performance drops more quickly (e.g., 8K for CX-6). Similar trends exist for the RDMA send latency as well as read and write verbs.

In contrast, RNICs' overall aggregated bandwidth with ScalaCN hardly drops. Compared to the default RCN settings with 15K offloaded flows, ScalaCN improves the average aggregated bandwidth of the RNIC by 40.4% and reduces the average packet forwarding latency by 30.5%. This is because ScalaCN proactively reorganizes the RNIC's flow tables when performance degradation is about to happen.

Flow Startup Delay. We measure the startup delay of new flows to show the impacts of ScalaCN on flow initiation with flow table reorganization. Figure 16 depicts the breakdown of the flow startup delay under different numbers of flows. As the RCN scale increases, the startup delay of a flow gradually increases. Nevertheless, most (70%) of the delay is attributed to the OVS, which determines packet forwarding rules in the user space. The additional delay incurred by ScalaCN merely accounts for 18%, which is close to that of the driver (12%). Since the startup delay only affects a couple of packets on flow creation, we feel that such an increase is acceptable given the significant performance improvements of ScalaCN.

Prediction Accuracy. We measure the prediction bias of ScalaCN on the RNIC's performance, and compare it with a baseline prediction method that uses mainstream machine learning techniques (e.g., SVM) with representative flow features like packet loss rate and the number of flows [44, 69]. As shown in Figure 17, ScalaCN achieves a high accuracy on predicting the RNIC's packet queue utilization, with the maximum bias being only +2.82%. Based on this, ScalaCN can further predict the bandwidth and latency with a high accuracy of 98.9% and 98.5%, respectively. In contrast, the baseline method presents poor accuracy, with the maximum bias being -53.98% and +76.76% for bandwidth and latency.

5.3 Real-world Production Workloads

Performance Benefits. We measure the bandwidth improvement and latency reduction on the RNICs that carry our production RCN workloads. As shown in Figure 18 and Figure 19, ScalaCN improves the bandwidth by 17% on an average RNIC, and reduces the average latency by 15%. Also, the maximum bandwidth improvements reach 43.9%, 37.9%, 41.8%, 28.5%, and 31.2% for CX-4/5/6/7, and BF-3, respectively. The latency reductions are 39.0%, 34.1%, 35.9%, 27.2%, and 26.9%, respectively. Similar results exist for the Intel device E810 (not shown due to the space limit).

Different RNIC models benefit differently from ScalaCN. The CX-4 RNIC which was released early benefits the most from ScalaCN. We assume that this is because the early design of RNICs has a longer delay for each flow table query due to their inferior hardware components. Thus, reducing the number of queries substantially improves efficiency. Note that for the more advanced CX-7 and BF-3 RNICs, ScalaCN still provides considerable performance improvements (i.e., 105 Gbps) since it resolves the hardware bottleneck and reduces the processing complexity for RCN packets.

On the other side, ScalaCN can induce minor performance drops in some cases, in particular the computation-intensive tasks that involve few data communications among containers. The performance drop mainly derives from the flow entry insertion to the hyper matching mask on frequent flow creation. Nevertheless, such a performance drop is trivial ($<5\%$) and only occurs to $<0.03\%$ RNICs in our RCN.

Overhead. ScalaCN needs to continuously monitor the crucial information from software and hardware stacks for performance optimization. As shown in Figure 20, ScalaCN incurs low CPU overheads on different scales of flow offloading. When the host carries more flows, ScalaCN needs to monitor more flow states, and therefore will incur higher CPU overhead. Such overhead is linear to the number of offloaded flows and will finally converge to $\sim5\%$ on a single core. In the real-world production RCN, the overhead is negligible given the usually tens of CPU cores running on a host.

6 Lessons Learned

Abstraction before Testing. For the performance testing of networking devices such as RNICs and switches, it would be more efficient if we prepare a general abstraction before the actual testing. Such an abstraction can be derived from the hardware datasheet, open-source drivers, and common network APIs. With this abstraction, we can efficiently infer the potential performance bottlenecks and the root causes of the performance issues inside the hardware of commodity devices. In particular, the abstraction helps us identify combinatorial restrictions of the device's input, so that the search space of the performance testing can be largely reduced.

Blackbox Hardware Is Practically Understandable. Although commodity RNICs are mostly close-sourced, we can still shed light on their basic architecture and performance models by concentrating on their critical packet processing pipeline. Based on this realistic insight, CSPs like us can take proactive measures (e.g., optimizing the flow offloading schedule) to mitigate unexpected issues of the RNICs, without the need to modify or re-architect the RNIC (which is costly and at least takes time). Even if we cannot accurately profile the behavior of an RNIC, we can validate some of its key metrics like bandwidth and latency to determine whether a performance degradation is within the acceptable range.

7 Related Work

RNIC Performance Enhancement. Over the past decade, RNICs have gone through a remarkable evolution. Many studies focus on enhancing the performance and scalability of RNICs [25, 29, 33, 59, 62, 63]. SRNIC [63] is a scalable architecture based on FPGA, which minimizes on-chip data structures. StaR [62] balances state maintenance between two communication ends to improve scalability. These studies mostly focus on addressing resource contentions (i.e., hardware limits) of the on-chip memory, while ScalaCN focuses on the on-chip packet processing pipeline that is rarely touched in previous work but is more important for the large-scale use of RNICs. Unlike hardware limits that require re-architecting the hardware, focusing on the packet processing pipeline provides us with an opportunity to enhance the performance even if we have limited visibility into the RNIC's internals.

Troubleshooting Data Center Networks. There have been plenty of studies on troubleshooting data center networks. Most of them focus on verifying the correctness of control-plane rules [17, 20, 23, 50, 61, 68]. Minesweeper [17] and Plankton [50] are two representative approaches to validating network configurations. However, these methods all assume that the underlying RNICs will not fail, and the states as well as performance statistics in the software stack always stay consistent. Such an assumption in fact does not always hold in the production RCN, since RNICs' capability limits can only be discovered in large-scale settings. In contrast, we dissect the RNICs' scalability walls in a large-scale production RCN. This enables us to design ScalaCN to practically address the performance issues when the network quickly scales up.

8 Conclusion

This work presents our efforts towards understanding and mitigating the scalability limit of RCN in a large-scale production environment. In particular, we leverage the efficient approach of *combinatorial causal testing* to interpret the performance issues of RNICs. The derived architecture and performance models guide us to reliably infer the bottlenecks inside RNICs and devise an effective method to overcome the bottlenecks. RNIC vendors' feedback validates our multifold findings and comprehensive evaluation results confirm the efficacy of our solution. In a broader sense, our work provides a principled approach to extracting the high-level mechanisms of proprietary hardware devices, which can benefit a wider range of fields such as hardware testing, verification, and security.

Acknowledgments

We acknowledge Container Network and High-Performance Network teams in Alibaba Cloud that contributed to the success of ScalaCN. We thank our shepherd, Mariano Scazzariello, and the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by the National Key R&D Program of China under grant 2022YFB4500703, the NSFC under grants 62332012 and 62472245, and the Alibaba Research Intern Program.

References

[1] Ethernet Switch Device Driver Model (switchdev) — The Linux Kernel Documentation. https://docs.kernel.org/networking/switchdev.html, 2015.

[2] Introducing Cloud Native Networking for Amazon ECS Containers | AWS Compute Blog. https://aws.amazon.com/blogs/compute/introducing-cloud-native-networking-for-ecs-containers/, 2017.

[3] Advanced Traffic Control - ArchWiki. https://wiki.archlinux.org/title/advanced_traffic_control, 2023.

[4] Configure Azure CNI Overlay networking in Azure Kubernetes Service (AKS) - Azure Kubernetes Service. https://learn.microsoft.com/en-us/azure/aks/azure-cni-overlay, 2023.

[5] Flow Hardware offload with Linux TC flower — Open vSwitch 3.3.90 documentation. https://docs.openvswitch.org/en/latest/howto/tc-offload/, 2023.

[6] tc-flower(8) - Linux Manual Page. https://man7.org/linux/man-pages/man8/tc-flower.8.html, 2023.

[7] Compare network models in GKE | Google Kubernetes Engine (GKE). https://cloud.google.com/kubernetes-engine/docs/concepts/gke-compare-network-models, 2024.

[8] Container Network Interface (CNI). https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/, 2024.

[9] Docker: Accelerated Container Application Development. https://www.docker.com/, 2024.

[10] Kubernetes. https://kubernetes.io/, 2024.

[11] LamaConv—Logics and Automata Converter Library. https://www.isp.uni-luebeck.de/lamaconv, 2024.

[12] NVIDIA Mellanox ConnectX-6 SmartNIC Adapter | NVIDIA. https://www.nvidia.com/en-sg/networking/ethernet/connectx-6/, 2024.

[13] Overview - Container Service for Kubernetes - Alibaba Cloud Documentation Center. https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/overview-18, 2024.

[14] perftest. https://github.com/linux-rdma/perftest, 2024.

[15] André Altmann, Laura Toloşi, Oliver Sander, and Thomas Lengauer. Permutation Importance: A Corrected Feature Importance Measure. *Bioinformatics*, 26(10):1340–1347, 2010.

[16] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):1–64, 2011.

[17] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proc. of ACM SIGCOMM*, 2017.

[18] Arthur Bernstein and Paul K. Harter. Proving Real-Time Properties of Programs with Temporal Logic. In *Proc. of ACM SOSP*, 1981.

[19] Emanuele Borgonovo and Elmar Plischke. Sensitivity Analysis: A Review of Recent Advances. *European Journal of Operational Research*, 248(3):869–887, 2016.

[20] Tobias Bühler, Romain Jacob, Ingmar Poese, and Laurent Vanbever. Enhancing Global Network Monitoring with Magnifier. In *Proc. of USENIX NSDI*, 2023.

[21] Martin Dietrich Buhmann. Radial Basis Functions. *Acta Numerica*, 9:1–38, 2000.

[22] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding Host Network Stack Overheads. In *Proc. of ACM SIGCOMM*, 2021.

[23] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE Way to Test OpenFlow Applications. In *Proc. of USENIX NSDI*, 2012.

[24] Sagun Chanillo and Michael Kiessling. Rotational Symmetry of Solutions of Some Nonlinear Problems in Statistical Mechanics and in Geometry. *Communications in Mathematical Physics*, 160(2):217–238, 1994.

[25] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proc. of ACM EuroSys*, 2019.

[26] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proc. of USENIX NSDI*, 2018.

[27] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marcaurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large Scale Distributed Deep Networks. In *Proc. of NIPS*, 2012.

[28] Yaozu Dong, Yu Chen, Zhenhao Pan, Jinquan Dai, and Yunhong Jiang. ReNIC: Architectural Extension to SR-IOV I/O Virtualization for Efficient Replication. *ACM Transactions on Architecture and Code Optimization*, 8(4), 2012.

[29] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA Over Commodity Ethernet at Scale. In *Proc. of ACM SIGCOMM*, 2016.

[30] Alexander Van't Hof and Jason Nieh. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proc. of USENIX OSDI*, 2022.

[31] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proc. of ACM SOSP*, 2023.

[32] Tao Ji, Divyanshu Saxena, Brent E Stephens, and Aditya Akella. Yama: Providing Performance Isolation for Black-Box Offloads. In *Proc. of ACM SoCC*, 2023.

[33] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proc. of USENIX OSDI*, 2016.

[34] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating Network-based CPU in Container Environments. In *Proc. of USENIX NSDI*, 2018.

[35] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-Based Virtual RDMA Networking for Containerized Clouds. In *Proc. of USENIX NSDI*, 2019.

[36] Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *Proc. of USENIX NSDI*, 2023.

[37] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *Proc. of USENIX NSDI*, 2022.

[38] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T. V. Lakshman. UNO: Uniflying Host and Smart NIC Offload for Flexible Packet Processing. In *Proc. of ACM SoCC*, 2017.

[39] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. SocksDirect: Datacenter Sockets Can Be Fast and Compatible. In *Proc. of ACM SIGCOMM*, 2019.

[40] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, Jianbo Dong, Lingbo Tang, Hongqiang Harry Liu, Shaozong Liu, Weijie Li, Rui Miao, Yaohui Wu, Zhiwu Wu, Chao Han, Lei Yan, Zheng Cao, Zhongjie Wu, Chen Tian, Guihai Chen, Dennis Cai, Jinbo Wu, Jiaji Zhu, Jiesheng Wu, and Jiwu Shu. Flor: An Open High Performance RDMA Framework Over Heterogeneous RNICs. In *Proc. of USENIX OSDI*, 2023.

[41] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-Tenant Networks. In *Proc. of USENIX OSDI*, 2020.

[42] Xu Liu, Peng Zhang, Hao Li, and Wenbing Sun. Modular Data Plane Verification for Compositional Networks. *Proceedings of the ACM on Networking*, 1(CoNEXT3), 2023.

[43] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM Is Lighter (and Safer) Than Your Container. In *Proc. of ACM SOSP*, 2017.

[44] Mariyam Mirza, Joel Sommers, Paul Barford, and Xiaojin Zhu. A Machine Learning Approach to TCP Throughput Prediction. In *Proc. of ACM SIGMETRICS*, 2007.

[45] Manfred Morari and Jay H Lee. Model Predictive Control: Past, Present and Future. *Computers & Chemical Engineering*, 23(4-5):667–682, 1999.

[46] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martın Casado. The Design and Implementation of Open vSwitch. In *Proc. of USENIX NSDI*, 2015.

[47] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. Autonomous NIC Offloads. In *Proc. of ACM ASPLOS*, 2021.

[48] Nikolai Pitaev, Matthias Falkner, Aris Leivadeas, and Ioannis Lambadaris. Characterizing the Performance of Concurrent Virtualized Network Functions with OVS-DPDK, fd.io VPP and SR-IOV. In *Proc. of ACM ICPE*, 2018.

[49] Amir Pnueli. The Temporal Logic of Programs. In *Proc. of IEEE FOCS*, 1977.

[50] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P Brighten Godfrey, and Matthew Caesar. Plankton: Scalable Network Configuration Verification Through Model Checking. In *Proc. of USENIX NSDI*, 2020.

[51] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. In *Proc. of USENIX OSDI*, 2020.

[52] Douglas A Reynolds et al. Gaussian Mixture Models. *Encyclopedia of Biometrics*, 741(659-663), 2009.

[53] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. Network-Centric Distributed Tracing with DeepFlow: Troubleshooting Your Microservices in Zero Code. In *Proc. of ACM SIGCOMM*, 2023.

[54] Jacopo Soldani and Antonio Brogi. Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey. *ACM Computing Surveys*, 55(3), 2022.

[55] Brent Stephens, Aditya Akella, and Michael M. Swift. Your Programmable NIC Should Be a Programmable Switch. In *Proc. of ACM HotNets*, 2018.

[56] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proc. of ACM ISCA*, 2023.

[57] Qiang Su, Chuanwen Wang, Zhixiong Niu, Ran Shu, Peng Cheng, Yongqiang Xiong, Dongsu Han, Chun Jason Xue, and Hong Xu. PipeDevice: A Hardware-Software Co-Design Approach to Intra-Host Container Communication. In *Proc. of ACM CoNEXT*, 2022.

[58] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An Analysis and Empirical Study of Container Networks. In *Proc. of IEEE INFOCOM*, 2018.

[59] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *Proc. of ACM SOSP*, 2017.

[60] Abraham Wald. *Sequential Analysis*. Courier Corporation, 2004.

[61] Weitao Wang, Xinyu Crystal Wu, Praveen Tammana, Ang Chen, and T S Eugene Ng. Closed-loop Network Performance Monitoring and Diagnosis with SpiderMon. In *Proc. of USENIX NSDI*, 2022.

[62] Xizheng Wang, Guo Chen, Xijin Yin, Huichen Dai, Bojie Li, Binzhang Fu, and Kun Tan. StaR: Breaking the Scalability Limit for RDMA. In *Proc. of IEEE ICNP*, 2021.

[63] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, Tianhao Wang, Weicheng Ling, Kejia Huo, Pingbo An, Kui Ji, Shideng Zhang, Bin Xu, Ruiqing Feng, Tao Ding, Kai Chen, and Chuanxiong Guo. SRNIC: A Scalable Architecture for RDMA NICs. In *Proc. of USENIX NSDI*, 2023.

[64] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *Proc. of USENIX NSDI*, 2023.

[65] Jian Yang, Joseph Izraelevitz, and Steven Swanson. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *Proc. of USENIX NSDI*, 2020.

[66] Xinlei Yang, Xianlong Wang, Zhenhua Li, Yunhao Liu, Feng Qian, Liangyi Gong, Rui Miao, and Tianyin Xu. Fast and Light Bandwidth Testing for Internet Users. In *Proc. of USENIX NSDI*, 2021.

[67] Zhuolong Yu, Bowen Su, Wei Bai, Shachar Raindel, Vladimir Braverman, and Xin Jin. Understanding the Micro-Behaviors of Hardware Offloaded Network Stacks with Lumina. In *Proc. of ACM SIGCOMM*, 2023.

[68] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *Proc. of USENIX NSDI*, 2014.

[69] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *Proc. of ACM SIGCOMM*, 2021.

[70] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *Proc. of USENIX NSDI*, 2019.

Appendix

A Design Details

State Validation. While the performance predictors enable ScalaCN to predict RNIC performance, we still need an effective approach to validating whether the RCN is running in the correct states to make up for possible prediction errors. We take advantage of the linear temporal logic (LTL) [18, 49] to model the expected behaviors RNICs according to the causal testing results. We choose LTL because it can describe the transitions of system states over time. It acts as a test oracle to validate the state correctness and actual performance of the RCN. For example, we use LTL policies "if a flow is offloaded, the RNIC should create a flow entry for it." and "a new flow entry should not be created until a flow is offloaded" to validate the states of RNIC flow entry creation. These two policies can be represented in LTL formulas [49] as $FL\_OFFLD \rightarrow \bigcirc FE\_CREATE$ and $\neg FL\_CREATE\ \mathbf{U}\ FE\_OFFLD$. ScalaCN encodes LTL policies to build the LTL validation automaton.

Figure 21 shows the example for the above two specifications. States S_0 and S_1 are undetermined states, S_2 is the error state (i.e., an RCN state inconsistency is detected), and S_3 is the accept state (i.e., the RCN stays correct). If a flow entry is created at the initial state S_0 (without flows offloaded), the automaton goes into the error state and indicates a state inconsistency. If a flow is offloaded to RNICs twice, ScalaCN also determines an inconsistency. Only when the OVS offloads a flow and the RNIC creates the corresponding flow entry, ScalaCN will determine that the RCN stays correct.

It is also possible that ScalaCN cannot identify any issues in the software and hardware stack, but the performance is still very poor (e.g., due to the hardware aging, erosion, or some other environmental factors). To handle these unexpected scenarios, we have a passive monitor on the container network besides ScalaCN. Such a monitor will notify the control plane immediately about the unexpected performance degradation, and the control plane will temporarily disable the allocations of these RNICs to the containers.

Flow Table Manipulations. The flow table policies are initially generated by the software switch. For example, when a new flow is created, it cannot be found in the flow table of an RNIC. In this condition, the RNIC cannot handle the new packet pattern and will transfer the processing of the packet to the software OVS, which will cause a lookup miss event [46] to the OVS. The OVS will then determine which port (i.e., VFs or VXLAN interfaces) this new packet should be sent to and whether it needs encapsulations or decapsulations. After processing the first packet of a new flow, the OVS also offloads the corresponding processing rules to the RNIC hardware. The key problem here is that the OVS is not aware of the RNIC's hardware architecture, and thus cannot offload the optimal flow rule schedule. In particular, it cannot well

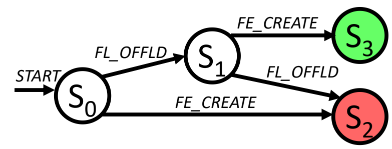


Figure 21: An example automaton from LTL specifications.

determine if the flow tables should be reorganized when the number of flow patterns increases. In contrast, ScalaCN can proactively transform these flow rules to the equivalent ones that minimize the overall flow processing delay for the RNIC.

Flow Table Locality. The concept of "locality" of a flow rule/table in ScalaCN means that it is frequently hit by the RCN packets. Matching masks with more bits set to "true" indicates that it can match more concrete flows if the table capacity is unlimited. However, it also occupies more flow table entries as each entry is more concrete (e.g., an extreme case is that the full-bit matching of the hyper mask requires each entry to match exactly one specific five-tuple). So the flow table is filled up more quickly for the hyper mask. Considering these two aspects, ScalaCN prioritizes the masks that have a wider range of pattern matching capability and a higher hit rate to the front of the flow table.

B Additional Evaluation Results

Convergence Speed of Causal Testing. ScalaCN completes the entire testing and discovers the performance bottlenecks for each of the tested RNIC models in 3.6 hours on average, which is 60× faster than the brute-force testing that goes through all the possible configurations in more than 9 days. Since our testing is based on the common abstractions of the RNICs in an RCN, the convergence speed for different RNICs is similar. These results demonstrate the effectiveness of our combinatorial causal testing approach in building a high-quality model for the RNICs in a greybox manner.

Impacts on Small Bursty Flows. As we have discussed in §4.3, the introduction of a hyper mask may cause a certain degree of performance degradation for small bursty flows, since their flow rules are not initially prioritized in the whole flow table. We conduct a set of experiments to evaluate the performance of ScalaCN on these small bursty flows, and find that the negative impacts are negligible compared with the performance without ScalaCN. This is because by default the OVS still needs to process the first packet of these small bursty flows without ScalaCN, which usually incurs the same high delay (e.g., tens of milliseconds) due to the lookup miss as in the case with ScalaCN. Also, the first packet of these small bursty flows has a higher probability to be processed by the first cascading mask in ScalaCN due to mask re-prioritizing. Thus, the first-packet latency is still acceptable (e.g., $<$5 us increase) with RNIC offloading. After the new flow is offloaded to the hyper mask (i.e., the second packet), such a performance degradation will be eliminated.