Transparent Multicore Scaling of Single-Threaded Network Functions

Lei Yan, Yueyang Pan, Diyu Zhou, George Candea, Sanidhya Kashyap
School of Computer & Communication Sciences
EPFL
Switzerland

Abstract

This paper presents NFOS, a programming model, runtime, and profiler for *productively* developing software network functions (NFs) that scale on multicore machines. Writing shared-state concurrent systems that are both correct and scalable is still a serious challenge, which is why NFOS insulates developers from writing concurrent code.

In the NFOS programming model, developers write their NF as a sequential program, concerning themselves with the NF logic instead of parallelism and shared-state synchronization. The NFOS abstractions are both familiar to the NF programmer and convey to the NFOS runtime crucial information that enables it to correctly execute the NF's packet processing in parallel on multiple cores. Paired with NFOS's domain-specific concurrent data structures, this parallelism scales the NF transparently, obviating the need for developers to write concurrent code. We show that serial, stateful NFs run atop NFOS achieve scalability on par with their concurrent, hand-optimized counterparts in Cisco VPP [8].

Some scalability bottlenecks are inherent to the NF's semantics, and thus cannot be resolved while preserving those semantics. NFOS identifies the root causes of such bottlenecks and provides scalability recipes that guide developers in *relaxing the NF's semantics* to eliminate these bottlenecks. We present examples where such NFOS-guided relaxation of NF semantics further improves scalability by 2× to 91×.

CCS Concepts: **• Computer systems organization → Multicore architectures**; **• Networks → Middle boxes / network appliances**; **• Software and its engineering → Multithreading**.

Keywords: Network functions, Transparent scaling, Concurrency, Performance profiling and debugging

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '24, April 22–25, 2024, Athens, Greece
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629591

ACM Reference Format:
Lei Yan, Yueyang Pan, Diyu Zhou, George Candea, Sanidhya Kashyap. 2024. Transparent Multicore Scaling of Single-Threaded Network Functions. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3629591

1 Introduction

The performance of software network functions (NFs) [13] is essential to today's Internet and data center networks, because NFs perform tasks that are on the critical path, such as firewalling, NAT, load balancing [22], or mobile core functions [29]. As user demand drives the line rate of modern NICs to hundreds of Gbps [9, 66], processing the network packets at these rates is challenging. For example, with a packet size of 64 bytes, to saturate a 100Gbps network port, an NF would have to process 1 packet every 6.7ns on average.

NFs that aim to scale to such workloads must process packets in parallel on many cores, because nanosecond-scale average processing rates with a single core are not feasible for real NFs on present or foreseeable CPUs. Not surprisingly, multicore parallelism has become the preferred architecture for both industrial NFs [5, 6, 8, 22] and academic ones [20, 30].

Unfortunately, despite decades of research, developing correct (i.e., free of concurrency bugs) and scalable (i.e., having a throughput that increases with the number of cores) concurrent programs remains a challenge [41, 55, 58], even for developers with extensive expertise. NFs turn out to be no exception. We conduct what is, to the best of our knowledge, the first study of NF concurrency bugs; we choose for our analysis the open source version of Cisco's Vector Packet Processing framework (VPP) [8], a mature and widely used, high-performance packet-processing stack that runs on commodity CPUs. We find that both scalability issues and concurrency bugs often affect the VPP NFs. Furthermore, identifying and fixing these bugs is non-trivial, usually taking months. Interestingly, VPP developers try to avoid writing concurrent code by partitioning the NF state among threads so as to form a shared-nothing architecture that is easier to reason about. Ironically, for several NFs, this approach requires surprisingly complicated coordination among threads, leading to bugs. Plus, the expected functionality of several NFs make it such that they cannot be implemented in a shared-nothing manner, because it imposes the need to share state between the cores that process packets.

Further work is needed to improve the productivity of developing scalable single-machine NFs. Most NF frameworks help developers scale NFs across *different* machines [24, 33, 49, 61] and/or simplify the programming of sequential NFs [21, 31, 37, 43, 47]. To our knowledge, the only framework for writing *single-machine* multicore NFs is Maestro [48], developed concurrently with NFOS. Maestro is limited to NFs that are amenable to exhaustive symbolic execution; for such NFs, Maestro produces shared-nothing parallel implementations when possible. Many NFs, however, impose the use of shared state as the price for preserving their expected semantics. In such cases, solutions that use coarse-grained locks to synchronize updates to shared state do not scale.

Generic techniques for making concurrent programming easier have serious limitations when applied to the NF domain (§2.1). Notably, automated loop parallelization techniques depend heavily on the independence of each loop iteration, which is uncommon in NFs. Concurrency bug detectors [14, 46, 57] do not address the challenge of fixing the found bugs. Systems that mask concurrency bugs during execution [38, 39, 42] suffer from high overhead and can only offer (partial) protection against specific types of concurrency bugs. Finally, current profilers cannot reveal the root causes of scalability bottlenecks [18, 19, 55, 60] nor provide effective and actionable suggestions to improve scalability.

We present NFOS, a programming framework that helps NF-domain experts develop scalable NFs without having to think about concurrency and parallelism. Our goal is to *improve their productivity.* The NFOS runtime and toolchain provide complete support for the programming, profiling, and optimizing of NFs. We address the challenges mentioned earlier by exploiting the domain-specific characteristics of NFs.

In the NFOS programming model, developers write their NFs as sequential programs. They get to use intuitive abstractions that hide concurrency, thus avoiding from the start the introduction of concurrency bugs. We expect this to improve productivity, shorten testing cycles, and speed up delivery of features and bug fixes. Unlike related work on automatic parallelization, discussed above and in §2.1, NFOS *parallelizes not the program but its processing of packets.* As an example of domain specificity, NFOS generalizes the notion of packet flows into the *packet set* abstraction, with which the developer groups logically connected packets.

The NFOS runtime parallelizes a sequential NF across available CPU cores *transparently* by leveraging the knowledge conveyed through the packet set abstraction. NFOS performs the processing of all packets in a packet set on the same core, exploiting the observation that NFs typically process packets by using state specific to those packets' flows [49]. For accessing shared state, NFOS does not use lock-based pessimistic synchronization but rather processes each packet in a memory transaction [28, 42], despite related work that claims such an approach to be unsuitable [48].

Processing packets in memory transactions is a good choice, beyond enabling transparent parallelization: First, processing a packet in the critical path is by necessity swift, leading to short critical sections that are a good match for transactional memory. Second, NFs employ a small, common set of data structures [64], with most concurrent accesses occurring in these structures. As a result, NFOS can tailor their implementation to be efficient in the context of transactional memory and to encapsulate common concurrency wisdom, such as fine-grained synchronization. Third, combined with the sequential programming model, NFOS can now eliminate altogether the possibility of NF developers introducing concurrency bugs like data races or atomicity violations.

Even though most NFs parallelized by NFOS achieve excellent scalability (as we show later), some NFs are inherently challenging, because their semantics impose frequent updates to state shared across all packets (e.g., NAT, MAC-learning bridge). To aid developers in quickly optimizing such NFs, NFOS includes a profiler and a set of *scalability recipes.* The profiler reports the number of transaction aborts caused by memory accesses on specific lines of code, which acts as a proxy metric for assessing the impact on scalability.

Unlike existing profilers, the NFOS profiler also identifies the *root causes* of scalability bottlenecks. Since NFOS directly manages concurrency, it can bridge the semantic gap between low-level aborts and application-level behavior, and can provide the profiler with a set of *conflict causes.* These encode the exact reasons for why transactions abort in the built-in concurrent data structures. For each conflict cause, NFOS provides recipes for improving scalability through relaxation of the NF's semantics, such as allowing for a read to have bounded staleness. Developers can easily follow these recipes without thinking of the intricacies of concurrency.

Our experimental evaluation shows that NFs developed with NFOS achieve scalability on par with functionally equivalent but manually parallelized and heavily optimized NFs from Cisco VPP [8]. The VPP optimizations include various lock-free algorithms and hardware-specific optimizations like prefetching. To illustrate the use of scalability recipes, we present three case studies of how, under the guidance of NFOS, one can productively identify and remove the root causes of scalability bottlenecks in NFs by relaxing semantics, with NF throughput improvement ranging from 2× to 91×.

In summary, this paper makes the following contributions:

- **NFOS abstractions and runtime** including domain-specific data structures that encapsulate concurrency best practices. We expect NFOS to improve NF developers' productivity by simultaneously freeing them from writing concurrent code and delivering competitive multicore performance.
- **Scalability recipes** that guide developers in quantitatively and systematically trading NF semantics for scalability. The NFOS profiler bridges the gap between low-level events and NF logic via conflict causes and the abort-rate proxy metric.
- **Study of NF concurrency bugs**, focused on VPP [59].

In the rest of the paper, we provide further background and motivation (§2), describe the target audience for NFOS and define a general model of NFs and their properties (§3), present the design and implementation of the NFOS framework (§4), conduct an experimental evaluation of NFOS (§5), and conclude with a discussion of limitations (§6).

2 Background and Motivation

Developing concurrent software—that is, software with multiple threads that access shared state—is challenging. In this section, we first discuss state-of-the-art techniques for making it easier to write concurrent code (§2.1). Then we present the state of practice in developing multithreaded NFs (§2.2). Finally, our study on concurrency issues in VPP NFs (§2.3) finds that, while some bug characteristics are similar to those in general server applications, others are specific to NFs.

2.1 Related work

NF programming frameworks. Existing NF programming frameworks provide abstractions to hide low-level details from programmers [31, 37, 43, 47]. For example, mOS [31] provides abstractions to hide flow processing. There are frameworks that help scale NFs across multiple machines in a distributed system setup [24, 33, 49, 61]. None of these are directly applicable to developing *multithreaded* single-machine shared-state NFs.

Automatic parallelization. Generic approaches for parallelizing sequential programs focus on extracting parallelism from loops and/or function calls [10, 26, 56]. These techniques depend on the independence of each loop iteration or function call, making them unsuitable for many NFs. For instance, in stateful NFs, processing of one packet depends on state changes resulting from processing previous packets. There exists work that converts sequentially accessed data structures into concurrent ones [16, 52, 62]. These approaches depend heavily on the semantics of the data structures being converted, and do not readily carry over to NFs.

Maestro [48], developed concurrently with NFOS, automatically generates a parallel version of an NF from its corresponding sequential code. Unlike NFOS, Maestro requires the NF code to be amenable to exhaustive symbolic execution. This requires that all loop bounds be static and that no pointer arithmetic be done outside certain data structures. For NFs whose semantics impose the use of shared state, such as a load balancer, Maestro pessimistically synchronizes access to all NF state using a global read-write lock, so every packet that triggers an update to NF state must wait to obtain exclusive access. Such coarse-grained synchronization hurts scalability of stateful NFs that perform updates frequently. For example, many NFs update packet counters on every packet arrival, and others update state for every new flow, which can happen frequently in the real world, where average flow size can be <20 packets [1].

Concurrency bug detection and masking. Prior research helps programmers find concurrency bugs [14, 46, 57]. This is valuable and practical, but does not offer NF developers a complete solution: even with effective detection, fixing concurrency bugs is still challenging [41]. There also exist approaches to masking concurrency bugs during program execution [38, 39, 42], but they only target a specific type of bugs and introduce non-trivial overhead, while not guaranteeing the absence of bugs. We do not expect developers of high-performance NFs to favor these approaches—they likely would prefer to not have concurrency bugs at all.

Identifying and fixing scalability bottlenecks. Besides correctness bugs, concurrent programming can also introduce performance bugs that lead to scalability bottlenecks. Existing profilers [18, 19, 60] identify the symptoms but not the root cause, which is what's ultimately needed for fixing the problem. For example, profilers can reveal symptoms like high contention on critical sections. Yet, such symptoms can be due to a variety of reasons: incorrect use of synchronization mechanisms (e.g., using exclusive locks instead of read-write locks for read-mostly data), improper synchronization granularity (e.g., coarse-grained locks to protect all shared state), etc. Existing profilers cannot tell which one is the root cause. Some recent work aims to provide guidelines for fixing scalability bottlenecks [55, 63], but they are typically too generic to be actionable for NF developers.

2.2 Current practice of developing concurrent NFs

We study the current practice of concurrent NF development by analyzing the source code, bug database, and mailing list for the NFs that form the Cisco VPP family [59], an open source yet commercial grade networking framework. While we cannot generalize beyond VPP, it does represent the most compelling group of high-performance NFs that we are aware of. We find that programmers employ multicore concurrency to make NFs scale, and they still use low-level synchronization primitives, such as atomic instructions and locks. This suggests that the limitations of existing programming frameworks and techniques discourage NF developers from adopting them. They may use certain tools, such as `perf`, to spot scalability bottlenecks and other concurrency bugs, but often fix them through trial and error. This appoach is error-prone, allows bugs to escape, and hurts productivity. We describe our findings in more detail below.

2.3 NF concurrency problems

Methodology. VPP [59] is a mature networking framework, developed over the course of 7 years. It is widely used and considered proven technology, as it helped ship over $1B of Cisco products [59]. We employ a methodology similar to prior work [41]: search the bug database for relevant keywords (e.g., "deadlock", "atomicity") and then manually

analyze randomly chosen bugs, to confirm that they are indeed concurrency bugs that manifested in NFs.

We formulate below four findings based on reading and analyzing these bugs. The first two match bug studies of generic server applications [41], while the last two are specific to NFs.

Bug type	**Number of bugs**
Data race	16
Atomicity violation	8
Deadlock	4
Total	28

Table 1. Concurrency bugs in VPP NFs.

Finding #1: Concurrency bugs are prevalent and harmful. Table 1 summarizes the concurrency bugs we found. Of the 30 concurrent NFs in VPP, our limited random bug search finds that at least 14 NFs have had concurrency bugs. Furthermore, all of the bugs we found are harmful: they cause the NF to produce incorrect output, to crash, or to hang.

Finding #2: Debugging concurrency bugs in NFs remains challenging. For the concurrency bugs we found, the time between introducing a concurrency bug and fixing it ranges from 1–75 months, with a median of 21 months.

Finding #3: Shared-nothing is not a panacea. To avoid the complexity of concurrency, a common design in NFs is to employ a shared-nothing architecture, where the NF state is partitioned among threads.

One bug [3] we found was in NAT's hairpinning mode, where the address mappings are partitioned among cores, and each thread owns a partition. The problem is that, when a thread processes a packet, it reads the address mapping of the packet's destination endpoint without synchronizing with the owner thread. Doing so leads to race conditions, such as the owner thread freeing the address mapping while another thread is still accessing it. The fix is to forward the packet to the thread that owns the address mapping of the packet's destination endpoint. This fix results in a massive 877 lines of changed code. Passing packets between threads to achieve correct shared-nothing appears in both the normal and hairpinning modes in VPP NAT. Unfortunately, as detailed below, it causes a scalability bottleneck.

Finding #4: Scalability bottlenecks are caused by simple concurrency patterns. We configure and run four VPP NFs: NAT, Firewall, and Load balancer on the CAIDA trace [1], and Bridge on a trace synthesized based on university data-center characteristics [12]. Of the four NFs, two do not scale with the number of cores and are far from reaching the 98 million-packets-per-second (mpps) limit of our traffic generator with one NIC port, as shown in Figure 1. For NAT, the scalability bottleneck turns out to be contention on queues that forward packets between cores. For Firewall,

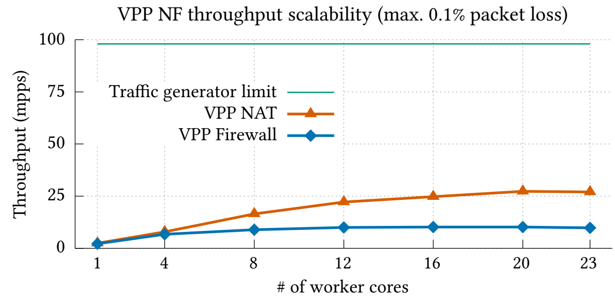

Figure 1. Throughput scalability of VPP NAT and Firewall.

the scalability bottleneck is caused by an allocator that inefficiently manages session table entries.

Conclusion. We believe that VPP developers, at least some of them, have extensive concurrency expertise—the code contains complex concurrency-related optimizations across many VPP NFs, including the use of lock-free algorithms. However, writing concurrent systems code is hard [41], as it requires non-intuitive, non-local reasoning about the interaction among threads. Concurrency bugs and scalability bottlenecks are hard to reproduce, as they can only be triggered by specific thread interleavings. Thus, even with concurrency expertise, NF developers can introduce concurrency problems and find it difficult to fix them correctly.

3 Target Users and NF Model

NFOS exploits several properties specific to the NF domain in order to overcome the challenges faced by related work (§2.1). To characterize the aspects of this domain that NFOS takes advantage of, we describe NFOS's target audience (§3.1) and present the model of NF operation assumed by NFOS (§3.2).

3.1 Target audience

NFOS targets developers with NF domain expertise. We assume that these developers understand the NF at the semantic level and, despite the subtleties of network protocols, these developers understand very well what are the correct outputs and state changes of an NF for given incoming packets. Furthermore, given the NF's semantics, the developers can implement such an NF, possibly using frameworks such as DPDK [21]. The developers have knowledge useful for optimizing NF performance, particularly which aspects of the NF's semantics are acceptable to trade for increased performance (e.g., they know whether it is acceptable for an NF to use specific stale data, if that improves scalability).

NFOS aims to help such NF domain experts, whether they have concurrency expertise or not. For the domain experts *without* concurrency expertise (that is, uncomfortable working with low-level synchronization primitives and implementing high-level synchronization algorithms), NFOS

enables them to develop functioning concurrent NFs with competitive multicore performance. For domain experts *with* concurrency expertise, NFOS helps quickly provide a baseline implementation. Only in the event that NFOS cannot provide satisfactory performance (this has not happened yet) do the experts need to write concurrent code themselves.

3.2 Target NF model

NFOS assumes that an NF operates in three stages: initialization, normal operation, and termination. During normal operation, an NF runs in an event loop. In each iteration, it receives a packet and refers to the NF's state to decide how to process that packet. To produce its output, an NF may modify, drop, or preserve the received packet, or generate an entirely new packet to send out. The NF also updates its state according to what it did. In the initialization (termination) stage, an NF creates (destroys) its state by allocating (freeing) the corresponding resources, respectively.

NFOS also assumes that NFs have the following properties:

P1: Flows mostly execute in isolated contexts. As noted in prior work [49], often an NF classifies a group of logically connected packets (e.g., packets belonging to the same TCP stream) into a "flow." When processing a packet, in the common case, an NF only reads or updates state that is *exclusively* associated with the flow to which the packet belongs. When state is semantically shared among flows, it is updated infrequently in the common case.

P2: Short packet-processing time. While a generic server application may take arbitrarily long to produce a response, NFs aim to process packets within a short amount of time, ranging from nanoseconds to at most microseconds. This is because delays in intermediate network nodes can significantly affect the end-to-end performance of the network.

P3: No ordering requirement on processing packets in different flows. An NF may require a specific packet processing order within a *single* flow. For example, packets in a TCP stream may need to be processed in their respective order. However, an NF typically does not require an ordering between processing packets in *different* flows. While different orders may result in different outputs, they are all valid.

P4: Concurrent accesses hit (mostly) a small set of widely used data structures. Most NFs maintain their state in a common set of data structures, like arrays, key-value maps, allocators, and/or counters [64]. For concurrent NFs, most of their concurrent accesses occur in these data structures, making their scalability critical to overall NF scalability.

P5: Limited use of the underlying system software. System software supporting generic applications offers a wide spectrum of functionality. For example, Linux and glibc provide hundreds of system calls and library methods. However, NFs use only a tiny fraction of this functionality.

We analyze the source code and descriptions of 14 widely used NFs (Table 6) and confirm that they match the model described here. 11 of the 14 NFs are organized around a certain type of flows. The reported average packet processing time varies from 90ns to 3µs, with one exception being the Web cache, which reports 50µs. We do not find any uses of synchronization primitives (e.g., conditional variables or barriers) for enforcing in-order packet processing across flows, nor do we believe it necessary given the NFs' semantics. These NFs indeed perform all their concurrent accesses inside the aforementioned data structures. Finally, they require from the underlying system only dynamic memory management, the current time, and disk/networking I/O.

Having described our target audience and the key properties of NFs, we now present NFOS's design.

4 The NFOS Framework

We now describe in detail how NFOS abstracts away concurrency and offers transparent scalability. We present NFOS's design goals (§4.1), an overview of the envisioned workflow (§4.2), the NFOS programming model (§4.3), the corresponding runtime (§4.4), the built-in data structures (§4.5), how the NFOS profiler and scalability recipes work (§4.6), select implementation details (§4.7), and an illustrative example of how to develop and optimize an NF with NFOS (§4.8).

4.1 Design goals (and non-goals)

The goal of NFOS is to help NF domain experts productively develop scalable NFs. We divide this overarching goal into the following four design objectives:

- **Intuitive abstractions to hide concurrency.** To enable high development velocity without a steep learning curve, the abstractions that NFOS uses to hide concurrency should be easily understandable with NF domain knowledge.
- **Encapsulated concurrency wisdom for competitive performance.** While we do not aim for concurrent NFs running on NFOS to always outperform hand-parallelized ones, we do want NFOS to be competitive for a wide range of NFs. To achieve this, NFOS should encapsulate general concurrency wisdom in its abstractions and runtime to maximize scalability.
- **Correct-by-construction concurrency.** Finding and fixing concurrency bugs is challenging, even for developers with extensive concurrency expertise (§2.3). Instead of providing support for detecting and fixing concurrency bugs, NFOS should *eliminate* the possibility of developers introducing concurrency bugs. However, preventing bugs that occur in sequential programming (e.g., buffer overflows) is out of scope.
- **Quick and easy scalability optimization.** It often takes many iterations of identifying and fixing scalability bottlenecks to achieve the desired performance. Thus, NFOS should include support for speeding up this process.

4.2 NFOS overview and workflow

NFOS exposes an event-driven, sequential programming model to NF developers. Figure 2 shows the workflow and key components, and Table 2 lists the event handlers that form the programming model. ❶ Developers program NFs by writing *sequential* NF code within these handlers (§4.3). ❷ The NFOS runtime then takes the sequential code and transparently makes it process packets on *multiple* cores (§4.4-§4.5). We describe later how we co-design the NFOS programming model and the runtime (i) to ensure that programmers cannot introduce concurrency bugs; and (ii) to extract fine-grained parallelism from the NF code to maximize scalability.

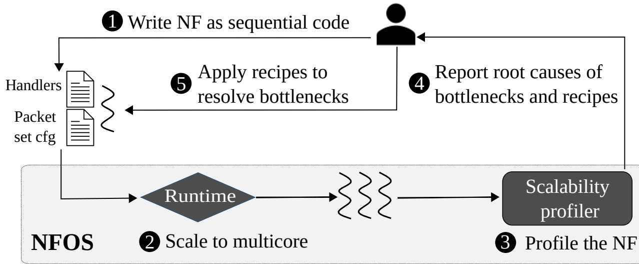

Figure 2. Typical workflow when using NFOS.

In most cases, NFs parallelized by NFOS achieve good scalability. However, NF semantics may impose inherent scalability bottlenecks, such as heavy contention on (implicitly) shared state. ❸ In this case, the developer uses the NFOS profiler (§4.6). ❹ The profiler directly reveals to the programmer the scalability bottleneck with an explanation of its root cause. For each possible root cause, NFOS provides the developer with one or several actionable *scalability recipes* to remove the bottleneck. The recipes are easy to apply, typically require changing at most a few lines of NF code, and guide the developer in making well thought-out and quantitative trade-offs between the NF's semantics and scalability (§4.6). ❺ The profiler and recipes help the developer productively optimize the NF's scalability (§4.6).

Figure 3 shows the NFOS system architecture:

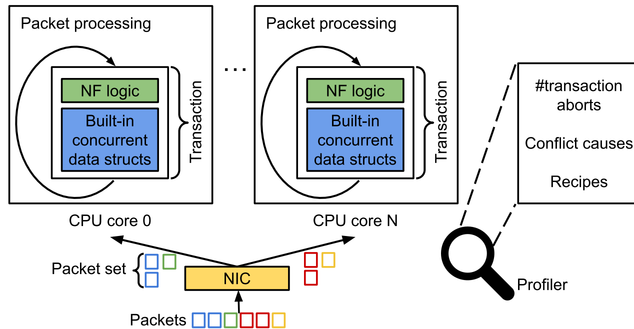

Figure 3. NFOS system architecture. Packets of the same color are logically related and belong to the same packet set.

Handler	When it is invoked
`init_NF_handler`	NF starts
`exit_NF_handler`	NF terminates
`init_pkt_set_handler`	The first packet of a packet set arrives
`pkt_handler`	A packet of a live packet set arrives
`expired_pkt_set_handler`	The packet set has expired
`orphan_pkt_handler`	An orphan packet arrives
`periodic_handler`	The recurring timer fires

Table 2. Handlers in the NFOS programming model.

4.3 NFOS programming model

The packet set is the unit of parallelism. NFOS must automatically identify any parallelism that exists in the sequential processing of packets, in order to transparently scale it. To do so, NFOS exploits NF property P1 (§3.2) to schedule the processing of different packets across multiple cores.

NFOS introduces the *packet set* abstraction to generalize the concept of network flow (§3.2). A packet set is a group of packets that are logically connected. Leveraging the observation that each flow forms an execution context (P1), NFOS minimizes synchronization overhead by processing all packets in a packet set on the same core, as further detailed below. Packets that do not belong to any packet set (which we call *orphan packets*) get evenly distributed across cores.

Programming with NFOS. A programmer using NFOS first defines packet sets in a configuration file by specifying the network protocols and the packet header fields that uniquely map a packet to a packet set (see Listing 1 for an example). The programmer then fills in the event handlers (Table 2), to implement the NF logic in plain C. Event handlers that the developer does not fill in remain no-ops.

We design the handlers to be generic enough to express most NFs. The first two handlers are used to initialize and clean up the NF, respectively. Similarly, there are handlers to initialize a packet set and clean it up. The initialization handler is invoked when the first packet of a packet set arrives. After the initialization, the packet set is "live," and `pkt_handler()` is invoked to process subsequent packets. The cleanup handler is invoked when a packet set expires (i.e., has not received any packet for a certain amount of time, specified during initialization). NFOS also has dedicated handlers for processing orphan packets and periodic events (e.g., to monitor the health of load balancer backends).

Programmers can use all of the standard C language when filling in the handlers. In addition, NFOS provides users with a set of library calls to support the basic NF functionality discussed in §3.2 and common data structures used by NFs (§4.5). However, using any concurrency-related variable types (e.g., `atomic`) or library calls (e.g., `pthread_*`) in the single-threaded NF is prohibited, to prevent the introduction of concurrency bugs.

Local vs. global state. In NFOS, NF state can be only of two kinds: (i) *packet-set state*, which is associated exclusively with a packet set and thus is only accessed by the NF when it processes packets in that packet set; or (ii) *aggregate state*, which is shared across packet sets, and thus may be accessed by the NF when it processes any packet. NFOS processes all packets in a packet set on the same core, so there is no need to synchronize access to the packet-set state—only access to the aggregate state needs to be synchronized.

To classify these two types of state, NFOS currently requires developers to annotate their variable definitions with two new types (`pkt_set_state` and `aggregate_state`), but in future versions we expect manual annotations to be obviated by tools like StateAlyzer [35]. The NFOS runtime instruments all accesses to the aggregate state so as to ensure correct and scalable parallelization, as detailed next.

4.4 Automatic scaling with NFOS runtime

Efficient and generic concurrency control with transactions. The mechanism that NFOS uses to synchronize access to aggregate state needs to be scalable. In addition, it needs to ensure that the concurrent NF performs as the developer expects, i.e., as in the sequential implementation, where each packet is processed atomically.

To that end, we chose *transactional memory* [28, 53] as the synchronization mechanism, from among several options that included lock inferences [40] and RCU [44]. While transactional memory cannot always achieve the best performance, it can achieve good performance under most scenarios by exploiting fine-grained parallelism. Specifically, two transactions can execute concurrently as long as they do not make conflicting accesses to the same object (i.e., no concurrent read-write or write-write accesses). More importantly, the behavior of transactional memory is simple and well understood: concurrent computation instances appear to execute sequentially and atomically, i.e., all-or-nothing semantics for operations within each instance. We defer the discussion of how NFOS ensures atomicity in the presence of events like I/O to the end of this subsection.

Following this design choice, NFOS executes each of its handlers in a transaction; the transaction starts at the beginning of a handler and commits when the handler returns. During the execution of the handler, NFOS tracks the accesses to aggregate state so that it can abort the transaction if there is a conflicting access to that aggregate state. In such a case, NFOS retries the aborted transaction (we discuss liveness in §6). We implement the NFOS transactions in software based on MVRLU [36], which is a synchronization framework with efficient multi-versioning. Multi-versioning allows concurrent accesses to different versions of the same object between read-only and read-write transactions without causing aborts.

Avoiding concurrency bugs. With NFOS, the sequential programming model paired with transactional access to shared state eliminates all developer-introduced concurrency bugs. NFs on NFOS will not experience deadlocks or order violations, because no locks or ordering primitives are required (P3 in §3.2) or allowed in NF code. Low-level data races and atomicity violations lead to a transaction abort; upon re-execution and commit, they will have been functionally masked. As shown in Table 1, all of the 28 bugs we found in VPP belong to the four bug types mentioned here.

Overcoming challenges in applying transactional memory. Transactional memory simplifies concurrent programming [28, 42] but faces several challenges in real-world applications [41, 58]. We describe how NFOS addresses these challenges below.

Synchronization mechanisms and I/O operations are ill-suited to transactional memory. The former pessimistically avoids conflicting accesses, but doing so manually is unnecessary for correctness: a transaction automatically aborts on such conflicts. The latter requires complicated mechanisms for undoing its effects upon transaction abort. To resolve this issue, our programming model prevents the use of synchronization primitives and uses buffered I/O: all the I/O requests in transactions are buffered in memory and are discarded (or performed) when the transaction aborts (or commits), respectively. NFOS ensures that all the code of an NF handler is inside one and only one transaction, because nested transactions and the interaction between code in and outside transactions leads to complexity and non-intuitive semantics.

Since NFOS's transactional memory is based on automatically instrumenting NF accesses to aggregate state, executing code outside the NF (e.g., making library or system calls) could lead to problems: the NFOS runtime is not able to track and thus revert such state changes (e.g., file descriptor state changes due to `seek()`). Fortunately, unlike generic applications, NFs only require a few features from the underlying system software (P5 in §3.2). NFOS thus implements the necessary functionality in its runtime and integrates it with transactional memory. Finally, generic applications may have long code regions that cannot be wrapped in a transaction, but NFs normally do not have this challenge (P2 in §3.2).

4.5 Efficient scaling with built-in data structures

NFOS provides a set of built-in concurrent data structures, leveraging the fact that most concurrent accesses in an NF go to a small set of common data structures (P4 in §3.2). NFOS embeds in them extensive expertise in writing high-performance concurrent code, in the form of advanced optimizations, including pre-allocation and replication.

Developers may use one or more built-in data structures within a handler by invoking the corresponding APIs. Since built-in data structures are called from a handler, transactional memory ensures safe access. Within a handler (and

thus within a transaction), operations across multiple data structures execute atomically. Conflicting accesses performed inside built-in data structures abort the transaction. We next detail each of the built-in data structures, namely `Vector`, `Allocator`, `Map`, and `DistributedObj`, focusing on their interfaces and the enhancements to maximize scalability. The library of built-in data structures can be easily extended by concurrency experts, should the need arise.

`Vector` is an array of elements, with a `read()` / `write()` interface. In a `Vector`, each element is aligned on the cache line size, to prevent false sharing. To minimize memory overhead, we keep memory-transaction metadata in the spare space created by the alignment constraints. The same alignment approach is used for the other data structures as well.

`Allocator` helps NFs allocate and manage resources. To meet the common needs of NFs, the `Allocator` associates an expiration time with each allocated object. The `Allocator` frees an object after its time expires, thus making every allocation be actually a lease [25]. In addition to an `allocate()`/ `free()` interface, the `Allocator` also provides a `refresh()` function that the owner of an allocated object uses to reset its expiration time.

To maximize scalability, our design makes each core maintain a local pool of free resources. When its local pool becomes exhausted, a core obtains free resources from another core, whichever has the most free resources. To facilitate object reclaim, each core maintains a list of allocated objects, sorted by the remaining expiration time in descending order. Each `refresh()` moves the object to the head of the list.

`Map` maintains a one-to-one key-value mapping, useful for data structures like the MAC table in a network bridge. The `Map` has a `get()` / `set()` interface. We implement the `Map` as a bucket-based hash table, which allows concurrent accesses to different buckets to proceed in parallel, and only conflicting accesses within the same bucket abort the transaction. During initialization, `Map` allows users to specify the hash function that decides which bucket a key is mapped to.

`DistributedObj` is for objects with intensive concurrent write and read accesses (e.g., global counters). A distributed object has a `read()` / `update()` interface. The distributed object stores a portion of the object in each core. `update()` directly modifies the local portion of the invoking core, thereby avoiding the contention. `read()` requires iterating and merging the local portion of each core with a merging function (provided by the NF developer during the initialization of the distributed object) to obtain the final result. To mitigate the read-write contention, we make the distributed object store the last merged value, the current staleness (i.e., the time elapsed since the last merge), and the maximum staleness (specified by the NF developer during the initialization of the distributed object). If the object's staleness is smaller than the maximum staleness, `read()` does not iterate but directly returns the last returned value. Thus, developers can reduce read-write contention by increasing the maximum allowed staleness.

4.6 The NFOS profiler and scalability recipes

Existing profilers rarely reveal directly the root causes of scalability bottlenecks, nor do they provide actionable guidelines to improve scalability (§2.1). This is due to the semantic gap between what the profiler sees and what the application does. The profiler can, at best, understand the semantics of low-level synchronization primitives (e.g., locks and atomic operations), but it cannot understand the high-level concurrent data structures and algorithms in the application, thereby failing to identify the root cause behind the scalability issue.

Since NFOS completely manages the NF's concurrency, it can bridge this semantic gap. It overcomes the challenge by having the runtime, specifically the built-in concurrent data structures, convey to the NFOS profiler key knowledge, such as the intended behavior of each interface. This enables the profiler to directly distill for developers the root causes and the possible fixes of the scalability bottlenecks.

Bridging the semantic gap with conflict causes. NFOS provides the *conflict causes* abstraction to bridge the profiler with the built-in data structures. Each built-in data structure comes with a set of conflict causes. A conflict cause specifies the root cause of a transaction abort. It takes the form of a condition between a pair of data-structure operations that, if met, leads to conflicting concurrent accesses that will abort the transaction. Table 3 shows the conflict causes for `Map`.

	`get(key2)`	`set(key2)`
`get(key1)`	No conflict	*Conflict cause*: key1 equals key2 *Recipe*: Use distributed objects *Conflict cause*: The two keys hash to the same bucket *Recipe*: Increase map size or change hash func
`set(key1)`		Same as `get(key1)` and `set(key2)`

Table 3. Conflict causes and recipes for `Map`. A <*row,col*> cell indicates the conflict causes and recipes in case of a conflict between *row* and *col*. A diagonal line indicates an operation pair already present in another cell.

Scalability recipes. Each conflict cause further comes with one or more recipes that guide developers in mitigating or eliminating the associated scalability bottleneck. The recipes "translate" for the developer what a concurrency expert would do to resolve that scalability bottleneck. Examples of recipes include over-provisioning resources, allowing stale reads, or using alternate built-in data structures.

Conflict causes provide programmers with the precise reason for a scalability bottleneck, and the associated recipes

	read()	update()
read()	No conflict	*Conflict cause*: Object exceeds maximum staleness (thus read merges local portions) *Recipe*: Increase maximum allowed staleness
update()		No conflict

Table 4. Conflict causes and recipes for `DistributedObj`.

	allocate()
allocate()	*Conflict cause*: A local pool becomes exhausted, causing contention on another pool *Recipe*: Overprovision resource

	refresh(res2)
refresh(res1)	*Conflict cause*: res1 and res2 placed on the same allocated list *Recipe*: Increase refresh interval

Table 5. Conflict causes and recipes for `Allocator` (we show only a subset, due to space limitations).

provide solutions for optimizing the bottleneck. Tables 3, 4, and 5 show the conflict causes and recipes for `Map`, `DistributedObj`, and `Allocator` respectively. For `Vector`, the conflict causes are justs concurrent read-write or write-write accesses to the same element, and the recipes are to replace contended elements with distributed objects.

The developer of a built-in concurrent data structure specifies its conflict causes and the recipes based on the specifics of its implementation. For example, since `Map` is a bucket-based hash table (§4.5), one conflict cause is that two `set()` calls contend on the same bucket. A recipe is thus either to increase the map size or to specify a different hash function for load balancing when initializing the `Map`.

We design the interfaces of built-in data structures so that most recipes can be applied by simply changing the parameters in the initialization function. Scalability recipes often involve relaxing the NF's semantics, i.e., relative to the NF's specification, they introduce additional behavior. Applying the recipe thus involves a trade-off between NF scalability and semantics. With NFOS, developers can use their domain-specific knowledge to identify the sweet spot.

NFOS scalability profiler. The profiler reports two kinds of information. First is the number of transaction aborts (and the fraction of total transaction aborts it represents) caused by each line of the serial NF code. The number of transaction aborts serves as a proxy metric for the scalability cost of shared-state contention. Second, the profiler provides a ranked list of conflict causes and recipes when transaction aborts occur inside built-in data structures. In this case, the profiler first collects which conflict causes are responsible for each of the aborts. Then it ranks conflict causes based on the number of transaction aborts they cause. For each conflict cause, the profiler also reports the corresponding recipe(s).

4.7 Implementation

We prototyped NFOS as a user-space C library on top of the DPDK kernel bypass [21], which NFOS uses for network I/O. Inside each transaction, to minimize overhead while correctly detecting conflicting accesses, NFOS tracks memory accesses at the object granularity (instead of, say, at the byte granularity). This NFOS implementation is publicly available at https://github.com/dslab-epfl/nfos.

Distributing packets with RSS. To minimize synchronization overhead, NFOS distributes packet sets among cores and achieves load balancing by leveraging the Receive-Side Scaling (RSS) feature [50] of network interface cards (NICs). RSS distributes packets among NIC receive queues, which are pinned 1:1 to the CPU cores, based on the value of a set of packet header fields. By configuring RSS to use all or a subset of the packet header fields specified in the packet set definition, NFOS guarantees that all packets of a packet set are directed by RSS to the same core at line rate.

NFOS includes an automated checker that tells developers whether their packet set definitions are compatible with the NIC's RSS. The RSS checker is meant to be part of the NF developers' toolchain, to be used both during design and during compilation. In the case of incompatibility, the checker reports the unsupported packet header fields. Developers can then choose to change the definition to avoid the unsupported header fields or not define packet sets at all. However, we have yet to encounter an incompatibility—the packet set definitions of all NFs in Table 6 are supported by mainstream 100Gbps NICs like Mellanox Connectx5 and Intel E810.

RSS does not guarantee optimal load balancing [11]. Nevertheless, for our workloads, we did not observe any impact on NF performance resulting from imbalance. If necessary, NFOS could adopt RSS++ [11] for better load balancing.

Transaction batching. Many NFs process individual packets quickly (e.g., 90ns in §5.3). Therefore, NFOS amortizes the overhead of starting and committing transactions by batching the NF handlers of multiple packets in a single transaction. The batch size equals the number of packets the DPDK device driver polls from the NIC in a single `recv()` operation, which is at most 32. Since a transaction aborts immediately upon encountering the first conflicting access in a packet batch, the profiler will not account for potential conflicting accesses caused by processing later packets in the batch. The profiler effectively samples conflicting accesses. However, this does not affect the accuracy of the profiler in identifying and ranking conflict causes, because the number of conflicting accesses seen by the profiler will still be significantly higher (in the millions) than the number of possible conflict causes.

NFOS profiler. For each invocation of the built-in data structure operations, the profiler keeps the arguments, the data structure's internal state, and the thread ID. Upon transaction abort, the profiler records the conflicting data structure operations. In our current implementation, after the NF finishes execution, the profiler invokes a function provided by the corresponding built-in data structure to map each abort into its conflict cause. To prevent results from being skewed by profiling, we minimize the profiler overhead with several optimizations. Most importantly, since NFOS's transactional memory is based on object multi-versioning [36], the profiler records only the timestamps of the conflicting data structure operations in the critical path, and then obtains the data structures' internal state off the critical path.

4.8 Developing scalable NFs with NFOS

We use a DDoS attack detector NF (Anti-DDoS) adapted from NFF-Go [7] to illustrate the process of developing scalable NFs with NFOS. Anti-DDoS checks whether >=80% of the incoming TCP flows have only one packet, in order to detect SYN flooding and similar attacks.

A packet set in Anti-DDoS consists of all TCP packets with the same source and destination IP and port (i.e., each TCP flow); the definition is shown in Listing 1.

```
/* Each TCP flow is a packet set.
 * NIC 0 receives the packets. */
[{nic: 0,
  pattern: {"ipv4":    [src_ip, dst_ip],
            "tcp": [src_port, dst_port]}
}]
```

Listing 1. Packet set definition of Anti-DDoS.

Listing 2 shows the code for Anti-DDoS. The aggregate state is a distributed object struct (`cntrs`) with two members (lines 4–6): `flows` records the total number of flows, and `onePktFlow` records the number of flows with only one packet. The `init_pkt_set_handler` (lines 13–24) handles the first packet of a TCP flow. It increments `flows`, temporarily counts the flow as a one-packet flow by incrementing `onePktFlows`, and then checks for a DDoS attack. Upon receiving subsequent packets of a flow, NFOS invokes the `pkt_handler` (lines 26–32) to decrement `onePktFlows`. Finally, when a flow expires, the `expired_pkt_set_handler` (lines 33–37) decrements `flows` and then checks if the flow had only one packet and, if so, decrements `onePktFlows`.

With the default semantics, Anti-DDoS always reads the latest value of the flow counters when checking for DDoS attacks. This does not scale well under a simulated attack with many small flows—see the green series in Figure 4(a).

Therefore, the developer uses the NFOS profiler and recipes to figure out how to fix Anti-DDoS's scalability. The profiler's output (Figure 4(b)) shows that 97.5% of the transactions abort. 100% of the aborts are due to the first conflict cause

```
1  #define CNTR_STALENESS_MS 0 // flow cntr staleness in ms
2  /* NF state definitons */
3  pkt_set_state bool isOnePktFlow;
4  aggregate_state distr_obj_t (CNTR_STALENESS_MS) cntrs {
5    int flows, onePktFlows;
6  };
7  /* Callbacks for merging flow counter partitions */
8  void cntrs_merge(merged, partition) {
9    merged->flows += partition->flows;
10   merged->onePktFlows += partition->onePktFlows;
11 }
12 /* Handle first packet of a flow */
13 int init_pkt_set_handler(pkt) {
14   isOnePktFlow = true;
15   distr_obj_update(cntrs, flows++, onePktFlows++);
16   // register the packet set to finish init
17   register_pkt_set();
18   // read flow counters and detect DDoS
19   cntrs_l = distr_obj_read(cntrs, cntrs_merge);
20   if (cntrs_l->onePktFlows > 0.8 * cntrs_l->flows)
21     handle_ddos(); // take whatever action is needed
22   else
23     send_pkt(pkt);
24 }
25 /* Handle subsequent packets of a flow */
26 int pkt_handler(pkt) {
27   if (isOnePktFlow) {
28     isOnePktFlow = false;
29     distr_obj_update(cntrs, onePktFlows--);
30   }
31   send_pkt(pkt);
32 }
33 int expired_pkt_set_handler() {
34   distr_obj_update(cntrs, flows--);
35   if (isOnePktFlow)
36     distr_obj_update(cntrs, onePktFlows--);
37 }
```

Listing 2. Anti-DDoS network function written with NFOS. Some obvious types and parameters are omitted for clarity.

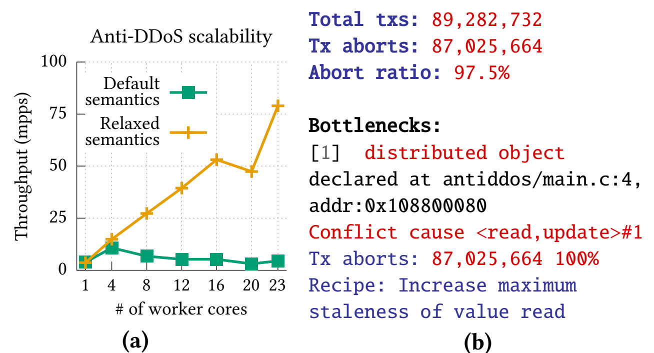

Figure 4. (a) Anti-DDoS throughput (max 0.1% packet loss) with default semantics (read latest flow counter value) vs. relaxed semantics (allowing for stale read with a maximum staleness of 0.1ms). **(b)** Profile of Anti-DDoS with default semantics.

of the concurrent read and update operations of distributed object `cntrs`. The profiler suggests increasing the maximum

allowed staleness of the distributed object. The developer applies this recipe by increasing `CNTR_STALENESS_MS` on line 1 from 0 to 0.1ms. The outcome is a negligible delay in detecting a DDoS attack, in exchange for improved scaling: semantic relaxation increases throughput by 18× at 23 cores, from 4.5 to 79 mpps—see Figure 4(a).

To summarize, motivated by the challenges of building concurrent NFs that scale on multicore machines, we propose the NFOS runtime and built-in data structures that offer NF developers a simple programming model. On top of that, NFOS also helps developers profile the scalability of their NF and offers actionable suggestions for how to improve it. We now present an experimental evaluation of our proposal.

5 Evaluation

We evaluate NFOS by answering the following questions:

- Does NFOS generalize to handle a wide range of NFs? Does it suitably abstract concurrency for all of them? (§5.2)
- How does the scalability of NFs written with NFOS compare to that of hand-parallelized NFs? (§5.3)
- Can developers easily improve the scalability of NFs using NFOS's profiler and recipes? (§5.4)

5.1 Evaluation setup

We use five NFs for our evaluation. We port to NFOS three NFs from Cisco's VPP [8] family of NFs: a MAC-learning bridge (Bridge), a load balancer (LB), and a network address translator (NAT). We compare the ported versions to the original, hand-parallelized versions. Bridge [17] forwards packets to a destination network based on the packet's MAC address and learns MAC-to-network mappings from each packet. LB implements the Maglev [22] algorithm. NAT is endpoint-independent [32, 54] and translates a LAN endpoint's IP and TCP/UDP port to the same public <IP:port> pair independently of the connecting WAN endpoint. We further implement with NFOS a stateful firewall (FW) that blocks external connections and a traffic policer (Policer) that rate-limits traffic by destination IP.

Our servers have two sockets equipped with 24-core Intel Xeon 6248R processors, 256 GB DRAM, and Intel E810 100Gbps NICs. All the NFs use DPDK 20.11, and we tune them for the best performance following DPDK's official suggestions [2]. We use MoonGen [23] to generate workload traffic; it transmits up to 98 million packets per second (mpps) per NIC port. The generator sends packets to the NFs on a different server and receives the processed packets from the NFs.

We stress the NFs using real Internet traffic from the CAIDA trace [1] and, for Bridge, using traffic synthesized based on a data-center trace [12]. The average TCP/UDP flow size for the CAIDA trace is 21 packets. For NAT and FW, we simulate LAN-to-WAN traffic with the CAIDA trace. For LB, since it only handles HTTP(S) connections, we modify the CAIDA trace to keep only the HTTP(S) packets and rewrite the destination IP of packets to LB's virtual IP. The synthetic trace for Bridge simulates two networks connecting through the NF. The trace has, for each pair of MACs and each direction, one flow with 10,000 packets, the same as the data-center average MAC flow size reported in [12].

5.2 Generality and ability to abstract concurrency

Generality. To assess whether NFOS's programming model is general enough to develop a wide range of NFs, we analyze 14 NFs used in industry and academia, shown in Table 6. We look at the code and semantics of each NF "by hand" and conclude that NFOS's programming model can express all the NFs. Defining packet sets is straightforward, as most of them end up being regular flows or sessions.

NF	Packet set	Packet-set state	Aggregate state
Endpoint-independent NAT [32, 54]	none	none	Public <IP:port> allocator; address mappings
Load balancer [22]	Incoming TCP flows	Flow-to-server mapping	Server health status
Bridge [17]	none	none	MAC-to-port mappings
DNStunnel detector [15]	none	none	Orphan DNS response counters
Sidejacking detector [15]	TCP session	Session context	Client IPs of each HTTP session
DDoS detector [7]	TCP/UDP flow	Flow size	Flow statistics
PRADS network monitor [61]	TCP/UDP session	Session context	Host info; traffic statistics
Web cache [4]	TCP session	Session context	Cached responses
Stateful firewall [64]	TCP/UDP session	Session firewall policy	none
Policer [45]	Packets to a LAN host	Incoming packet rate of the host	none
Portscan detector [51]	Packets from a host	Maliciousness probability	none
IPSec tunnel [34]	Tunnel session	Encryption key	none
nDPI deep packet inspector [31]	TCP/UDP session	Application protocol	none
Abacus [31]	TCP session	Data buffer	none

Table 6. Popular NFs used in industry and academia.

Abstracting away concurrency. We use the three concurrent NFs we ported to NFOS from VPP (Bridge, LB, NAT) to study whether NFOS can suitably mask concurrency from developers. The concurrency patterns in the VPP NFs are: (i) state partitioning among cores (flow–server mappings in LB, address mappings and public <IP:port> pairs in NAT) and (ii) concurrent data structures (lock-free MAC table in Bridge, server table in LB, and lock-free queues in NAT).

For all the ported NFs, NFOS completely ellimінates the need to handle the above concurrency patterns. Instead, developers only need to add, in total, 35–58 lines of code to: specify packet sets (0–5 LOC), categorize state (7–15 LOC), use NFOS built-in data structures (21–40 LOC), and fill in the handlers (7–16 LOC). The added code represents, at most, a few percentage points of the LOC implementing the NF logic.

We conclude that the NFOS programming model is sufficiently general to express the semantics of widely used NFs, is sufficiently familiar to NF developers to be used productively (e.g., packet sets are easy to define), and can fully abstract away the concurrency of the NF.

5.3 NF scalability, overhead, and latency

Figure 5 shows the throughput of five NFs written in NFOS. For the three we ported from VPP (Bridge, LB, and NAT), the semantics of the NFOS versions is the same as their original counterparts, so we can fairly compare to the original VPP NFs' throughput. The MAC table entry in Bridge uses a refresh interval of 1 minute, and NAT uses 57 public IPs. With our traces and NF setup, NAT and Bridge update aggregate state every 10 and 5,000 packets, respectively, while the other NFs have rare or no updates to aggregate state.

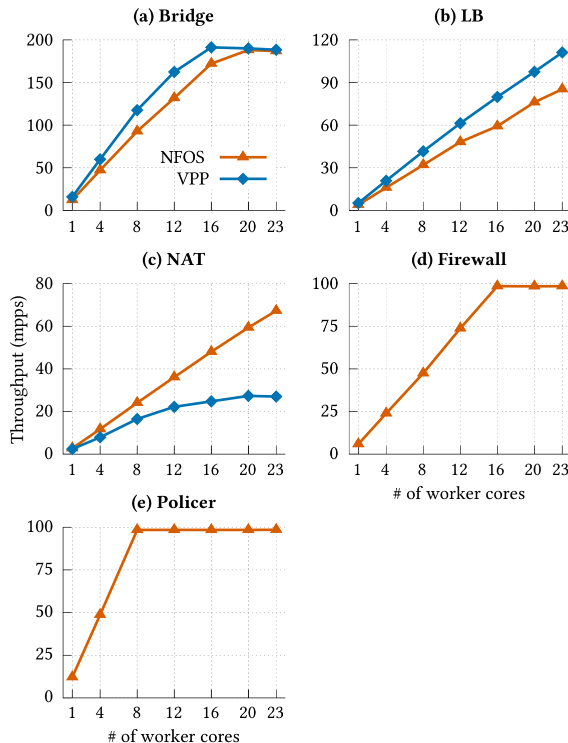

Figure 5. Throughput scalability with <0.1% packet loss. The maximum number of worker cores is 23 because both VPP and NFOS dedicate 1 core to auxiliary tasks (e.g., monitoring). Bridge, LB, and NAT were ported from VPP; Firewall and Policer are NFOS-only.

The NFOS NFs scale linearly, and three of the five reach the limit of the traffic generator or the servers' network I/O: 98 mpps for Firewall and Policer using one NIC port, and 188.8 mpps for Bridge using two NIC ports. The NFOS NFs' linear scalability is on-par with or better than the VPP NFs'.

In terms of absolute throughput, the NFOS NAT outperforms its VPP counterpart by up to 2.5×, NFOS Bridge trails VPP at low core counts but catches up at 23 cores, and the NFOS LB trails VPP by up to 25.88%, but would still saturate the network if it had a few extra cores. VPP NAT cannot scale because it suffers from contention on the shared queues for forwarding packets of a LAN endpoint to the core that needs to process them, which is a cost imposed by the partitioning of address mappings across cores. The NFOS NAT does not partition address mappings. For Bridge, the use of transactional memory accounts for all the overhead relative to VPP. For LB, it accounts for almost half. The rest of NFOS LB's overhead is due to cache misses: it uses a <src IP, src port, dst IP, dst port, L4 proto> 5-tuple as a key in flow-to-server mappings, while VPP LB uses a 32-bit hash, which saves 9 bytes per mapping. Given that the table has 5.8M entries, the space savings give VPP a higher cache hit rate.

These results confirm the benefit of the NFOS design that emphasizes linear scaling over absolute throughput: should the need for more throughput arise, instead of spending months optimizing performance in an NF-specific way, one can simply add a few more cores. As we will see shortly, this ease of developing and operating a scalable NFOS NF does not come at the price of higher latency.

Microbenchmarks. We use microbenchmarks to study how NFOS behaves under more targeted workloads. We develop a dummy NF in NFOS, whose aggregate state is an array with 10,000 elements. We simulate different workloads by varying the access type (RO=read-only, RW=read and write for each packet) and the skewness of accesses to the array (Zipfian values of 0 and 0.99).

Figure 6(a) shows the result. As expected, NFOS scales linearly with read-only workloads and reaches the limit of the traffic generator at 20 cores. This is because concurrent reads do not abort transactions. For the RW workload, NFOS scales well as long as the contention on the aggregate state is relatively low. The worst case for NFOS is the RW workload with high skewness. Fortunately, our experience with real-world NFs suggests that this situation is not at all common. Furthermore, in such a case, as detailed in §5.4, NFOS can guide developers to identify opportunities for improving scalability through semantic relaxation.

Handler batching. As mentioned in §4.7, NFOS batches the handlers of multiple packets into the same transaction, to amortize the overhead of transaction start and commit. Figure 6(b) shows the average speedup of batching over no-batching, varying the number of worker cores from 4 to 23. On average, batching improves the NF throughput by 34%.

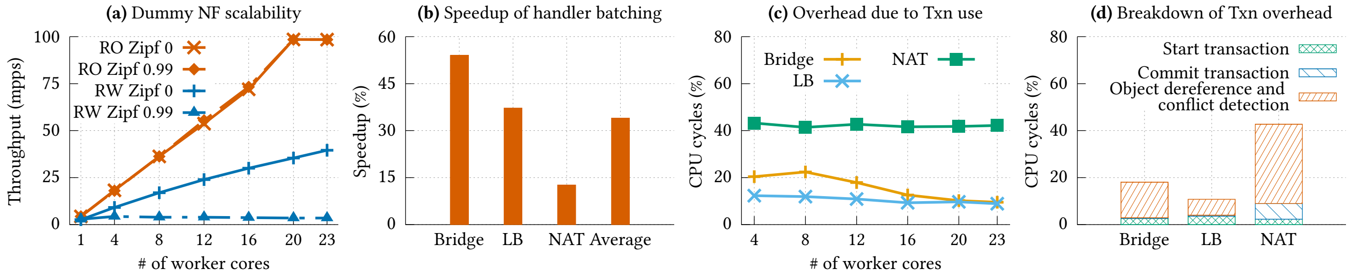

Figure 6. **(a)** Throughput (max 0.1% packet loss) of a dummy NF with varying aggregate-state access patterns. **(b)** Average NF performance speedup with batching. **(c)** Transactional memory overhead (measured as the percentage of CPU cycles spent in the transaction runtime). **(d)** Breakdown of transactional memory overhead.

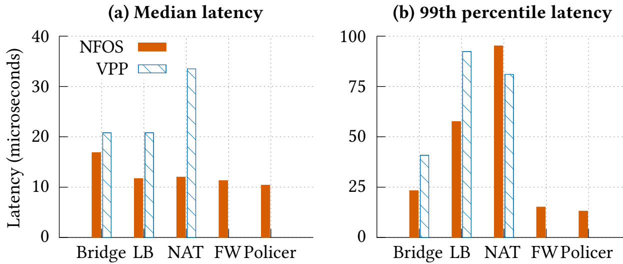

Figure 7. End-to-end packet latency for the five NFs.

Cost of using transactional memory to abstract away concurrency. We quantify this cost as the percentage of total CPU cycles spent in the transactional memory runtime, as reported by `perf`—see Figure 6(c). The transaction overheads are nearly independent of the number of cores, explaining the linear scalability of the evaluated NFs. Bridge's observed transaction overhead reduces after 12 cores because the NF reaches the limit of the testing server and is thus underloaded. Transactions do not add overhead to the Firewall and Policer, because these NFs do not have aggregate state. The synchronization overhead in hand-optimized concurrent NFs can be better or (a lot) worse, as shown in Figure 5.

Figure 6(d) shows the breakdown of the transaction overheads with 12 cores. Since NFOS's transactional memory is based on multi-versioning, most of the overhead comes from object dereferencing (i.e., obtaining the right version of objects) and detecting conflicting accesses.

Latency. Figure 7 shows the end-to-end packet latency (i.e., the latency between when the NF receives a packet at the NIC and when it sends the packet out on the wire) for the evaluated NFs with 23 worker cores. All NFs operate at the maximum throughput they can reach with less than 0.1% packet loss. VPP NFs have higher latencies in general due to aggressive packet batching.

Memory overhead. We analyze the memory footprint of NFOS NFs by measuring the maximum resident set size during execution. We find that the memory footprint of NFOS NFs increases only slightly as the number of cores increases from 1 to 23, with the maximum increase being 19% (Bridge: 39.5 to 47.1 MB; LB: 27 to 28.4 MB; NAT: 3,713.9 to 3,726.2 MB; Firewall: 25.5 to 26.1 MB; Policer: 24.8 to 25.6 MB). This modest increase comes from the per-core logs used in the NFOS transactional memory framework.

In summary, the throughput and latency of NFs developed with NFOS is on par with production-grade hand-parallelized NFs. We conclude that the ease of development, improved productivity, and avoidance of concurrency bugs offered by NFOS do not come at the price of seriously reduced performance.

5.4 Improving NF scalability with NFOS

Subsection 4.8 already presented one case study of improving NF scalability with NFOS. This subsection presents two more case studies with NAT and Bridge. Each MAC table entry (i.e., MAC-to-network mapping) in Bridge has a validity duration of 2 minutes, after which Bridge deletes the entry. Bridge can refresh a MAC table entry to reset its validity countdown.

We refer to the semantics of a sequential implementation of these NFs as the "default semantics": NAT uses 53 public IPs, and Bridge removes a MAC table entry only if it has not received a packet from the corresponding MAC for longer than the validity time interval. While such semantics are fine for a single-core execution, they make NFs inherently difficult to scale to multicore execution, as explained below.

NAT. Figure 8(a) shows that NAT with the default semantics stops scaling after 16 cores. We profile NAT with the NFOS profiler at 23 cores. The profiler shows that 99.4% of the transaction aborts are due to concurrent allocations (Table 5) made by an allocator that manages free public <IP:port> pairs. The profiler's recipe suggests overprovisioning the number of public <IP:port> pairs. We follow the recipe. As shown in Figure 8(a), NAT scales linearly with 55 public IPs, increasing throughput by 2× at 23 cores. Although public IPs are precious resources, under performance-critical scenarios, we believe that overprovisioning by <4% would be worthwhile given the significant scalability improvement.

Bridge. Figure 8(b) shows that Bridge cannot scale with the default semantics, because it has to refresh a MAC table entry each time it receives a packet from the corresponding

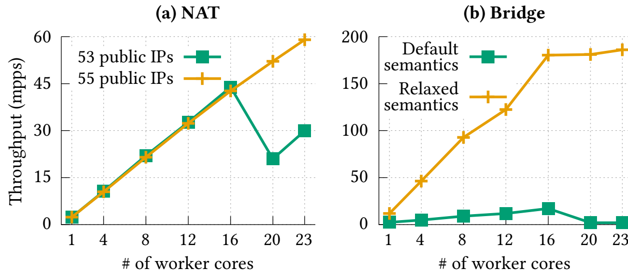

Figure 8. (a) NAT throughput (max 0.1% packet loss) with different numbers of public IPs. **(b)** Bridge throughput (max 0.1% packet loss) with the default semantics (MAC table entry refreshed on each packet) vs. the relaxed semantics (refresh interval 1 sec).

MAC address. The NFOS profiler shows that, with 23 cores, 63.5% of the transactions abort. Furthermore, almost all the aborts are due to concurrent refreshing of MAC table entries. We follow the NFOS-suggested recipe and increase the refresh interval from 0 to 1 sec (i.e., Bridge will only refresh a MAC table entry if it has not been refreshed for more than 1 sec). Figure 8(b) shows that this change makes Bridge scale linearly until it hits the network I/O limit. At 23 cores, its throughput increases by 91× (from 2.04 to 186 mpps).

Increasing the refresh interval relaxes Bridge's semantics in the sense that the MAC table entries could be removed (up to 1 sec) earlier than expected, potentially leading to some extra packet floods. For most cases this would be acceptable, given the improvement in common-case scalability and the infrequent packet flooding. The new refresh interval (1 sec) is still much smaller than the validity duration (2 minutes).

Summary. Our evaluation shows that NFOS can help developers easily write scalable NFs, even when the semantics pose inherent scalability challenges. The NFOS profiler shows the root causes of scalability bottlenecks, and then suggests to NF developers corresponding scalability recipes. Applying recipes requires changing only a few lines of NF code. Under NFOS's guidance, developers can use their domain-specific knowledge to make the right trade-off between scalability and NF semantics.

6 Discussion

Packet set definition. The packet set abstraction introduces a trade-off between load balancing and shared-state contention. Fine-grained packet sets (i.e., consisting of a small number of packets) can achieve good load balancing among cores but may require large amounts of aggregate state. This, in turn, may increase the contention on the aggregate state and thus limit NF scalability. Coarser-grained packet sets minimize the aggregate state but may incur load imbalance. This is because the number of packets in each packet set may differ substantially, so a few large packet sets could saturate a core while the other cores are left idle. Such a load imbalance may again limit performance.

For the NFs we studied (§5.2), we find that packet sets follow naturally from the NF's logic, and the packet sets we defined did not encounter the aforementioned problems. Nevertheless, we extended the NFOS profiler to also report the load imbalance among cores, to help developers choose the right packet set definition, should the need arise.

Starvation freedom. The current transactional memory implementation does not guarantee starvation freedom, so a thread could theoretically starve if it accessed highly contended aggregate state objects. Fortunately, our experience suggests that such high contention is uncommon in NFs (§5.2), and developers can avoid this issue by relaxing NF semantics (§4.6). As part of future work, we intend to investigate existing solutions to starvation in transactional memory [65].

NF chaining. Given the current NFOS design, operators can chain NFs parallelized by NFOS by executing each NF (and the associated NFOS runtime) in a virtual machine or container, and connecting the NFs with virtual switches [27]. However, this approach suffers from the overheads imposed by virtualization [47] and inter-core packet transfer. An interesting future direction would be to extend NFOS with NF chaining as a first-class design goal.

7 Conclusion

We presented a system that helps NF domain experts (with our without concurrency expertise) productively develop, profile, and optimize NFs that scale on multicore machines. NFOS enables developers to simply write NFs as sequential programs using NFOS data structures to store their state, as did Vigor [64]. This simplifies NF development and eliminates concurrency bugs "by construction." Developers use the intuitive *packet set* abstraction to indirectly convey to NFOS which packets can be processed in parallel and which cannot. The NFOS runtime then uses this information in combination with transactional memory to parallelize the processing of packets. The NFOS profiler bridges the semantic gap between low-level events and application-level concurrency and, by using the *conflict cause* abstraction, it reveals to developers the root causes of scalability bottlenecks. When these are inherent to the NF's semantics, the NFOS *scalability recipes* guide developers in trading NF semantics for scalability. Our experimental evaluation demonstrates that NFs written in NFOS achieve scalability that is similar to or better than that of manually parallelized NFs.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd, Jia-Ju Bai, for their constructive help in improving this paper. We are grateful to the Artifact Evaluation reviewers for their feedback on improving NFOS 1.0. We thank Katerina Argyraki, Solal Pirelli, Rishabh Iyer, Can Cebeci, Arseniy Zaostrovnykh, Georgia Fragkouli, Samuel Chassot, and Cristian Safta for providing feedback on the ideas, paper, and code.

References

[1] The CAIDA UCSD Anonymized Internet Traces - 2016. https://www.caida.org/catalog/datasets/passive_dataset. [Last accessed on 2023-10-29].

[2] DPDK Release 20.11. https://doc.dpdk.org/guides-20.11/rel_notes/release_20_11.html. [Last accessed on 2023-10-29].

[3] Fix of VPP NAT Race Condition on Address Mappings. https://gerrit.fd.io/r/c/vpp/+/31174. [Last accessed on 2023-10-29].

[4] HTTP Caching. https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching. [Last accessed on 2023-10-29].

[5] Juniper Networks vSRX Virtual Firewall Datasheet. https://www.juniper.net/us/en/products/security/srx-series/vsrx-virtual-firewall-datasheet.html. [Last accessed on 2023-10-29].

[6] netElastic Systems Carrier Grade NAT (CGNAT). https://netelastic.com/products/carrier-grade-nat-cgnat/. [Last accessed on 2023-10-29].

[7] NFF-Go. https://github.com/aregm/nff-go. [Last accessed on 2023-10-29].

[8] Vector Packet Processiong (VPP). https://github.com/FDio/vpp/tree/v21.01. [Last accessed on 2023-10-29].

[9] The Year of 100GbE in Data Center Networks. https://www.datacenterknowledge.com/networks/year-100gbe-data-center-networks. [Last accessed on 2023-10-29].

[10] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic Program Parallelization. *Proceedings of the IEEE*, 81(2), 1993.

[11] Tom Barbette, Georgios P Katsikas, Gerald Q Maguire Jr, and Dejan Kostić. RSS++: Load and State-Aware Receive Side Scaling. In *Intl. Conf. on Emerging Networking Experiments and Technologies (CoNEXT)*, 2019.

[12] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *ACM Internet Measurement Conf. (IMC)*, 2010.

[13] Lusheng Ji Bo Han, Vijay Gopalakrishnan and Seungjoon Lee. Network Function Virtualization: Challenges and Opportunities for Innovations. *IEEE Communications Magazine*, 53, 2015.

[14] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional Detection of Data Races. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, 2010.

[15] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A Declarative Language for Streaming Network Traffic Analysis. In *USENIX Security Symp.*, 2012.

[16] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K Aguilera. Black-Box Concurrent Data Structures for NUMA Architectures. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[17] LAN/MAN Standards Committee. IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks. *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)*, 2018.

[18] Charlie Curtsinger and Emery D Berger. Coz: Finding Code that Counts with Causal Profiling. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2015.

[19] Arnaldo Carvalho de Melo. The New Linux Perf Tools. http://vger.kernel.org/~acme/perf/lk2010-perf-paper.pdf. [Last accessed on 2023-10-29].

[20] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2009.

[21] DPDK: Data Plane Development Kit. https://dpdk.org. [Last accessed on 2023-10-29].

[22] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Symp. on Networked Systems Design and Implementation (NSDI)*, 2016.

[23] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM Internet Measurement Conf. (IMC)*, 2015.

[24] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. *ACM SIGCOMM Computer Communication Review*, 44(4), 2014.

[25] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *ACM Symp. on Operating Systems Principles (SOSP)*, 1989.

[26] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. Automatic Parallelization of Recursive Procedures. *Intl. Journal of Parallel Programming*, 28, 2000.

[27] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, 2015.

[28] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Intl. Symp. on Computer Architecture (ISCA)*, 1993.

[29] Evolved Packet Core (EPC) for Communications Service Providers. https://networkbuilders.intel.com/docs/networkbuilders/Evolved-packet-core-EPC-for-communications-service-providers-ra.pdf. [Last accessed on 2023-10-29].

[30] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: A Highly-Scalable Software-Based Intrusion Detection System. In *ACM Conf. on Computer and Communications Security (CCS)*, 2012.

[31] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Symp. on Networked Systems Design and Implementation (NSDI)*, 2017.

[32] Cullen Jennings and Francois Audet. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787, Internet Engineering Task Force, 2007.

[33] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. Stateless Network Functions. In *ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015.

[34] Charlie Kaufman, Paul Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, Internet Engineering Task Force, 2014.

[35] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *Symp. on Networked Systems Design and Implementation (NSDI)*, 2016.

[36] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[37] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3), 2000.

[38] Bohuslav Krena, Zdenek Letko, Rachel Tzoref, Shmuel Ur, and Tomás Vojnar. Healing Data Races On-the-Fly. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2007.

[39] Zdenek Letko, Tomás Vojnar, and Bohuslav Krena. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, 2008.

[40] Guangpu Li, Dongjie Chen, Shan Lu, Madanlal Musuvathi, and Suman Nath. SherLock: Unsupervised Synchronization-Operation Inference.

In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[41] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[42] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Intl. Symp. on Computer Architecture (ISCA)*, 2008.

[43] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Symp. on Networked Systems Design and Implementation (NSDI)*, 2014.

[44] Paul E McKenney and John D Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, 1998.

[45] Moonpol. https://github.com/erkinkirdan/moonpol. [Last accessed on 2023-10-29].

[46] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)*, 2007.

[47] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Symp. on Operating Systems Design and Implementation (OSDI)*, 2016.

[48] Francisco Pereira, Fernando M. V. Ramos, and Luis Pedrosa. Automatic Parallelization of Software Network Functions. In *Symp. on Networked Systems Design and Implementation (NSDI)*, 2024.

[49] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Symp. on Networked Systems Design and Implementation (NSDI)*, 2013.

[50] Introduction to Receive Side Scaling. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling. [Last accessed on 2023-10-29].

[51] Stuart E Schechter, Jaeyeon Jung, and Arthur W Berger. Fast Detection of Scanning Worm Infections. In *Recent Advances in Intrusion Detection*, 2004.

[52] Tomer Shanny and Adam Morrison. Occualizer: Optimistic Concurrent Search Trees From Sequential Code. In *Symp. on Operating Systems Design and Implementation (OSDI)*, 2022.

[53] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Symp. on Principles of Distributed Computing*, 1995.

[54] Pyda Srisuresh and Kjeld B. Egevang. Traditional IP Network Address Translator. RFC 3022, Internet Engineering Task Force, 2001.

[55] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *ACM EuroSys European Conf. on Computer Systems (EUROSYS)*, 2017.

[56] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2010.

[57] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and Surviving Data Races Using Complementary Schedules. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2011.

[58] Haris Volos, Andres Jaan Tack, Michael M. Swift, and Shan Lu. Applying Transactional Memory to Concurrency Bugs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[59] The Vector Packet Processing (VPP) Platform. https://wiki.fd.io/view/VPP/What_is_VPP%3f. [Last accessed on 2023-10-29].

[60] Intel VTune Performance Analyzer. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. [Last accessed on 2023-10-29].

[61] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *Symp. on Networked Systems Design and Implementation (NSDI)*, 2018.

[62] Zhengming Yi, Yiping Yao, and Kai Chen. A Universal Construction to Implement Concurrent Data Structure for NUMA-Muticore. In *Intl. Conf. on Parallel Processing*, 2021.

[63] Tingting Yu and Michael Pradel. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *Intl. Symp. on Software Testing and Analysis (ISSTA)*, 2016.

[64] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh R. Iyer, Matteo Rizzo, Luis Pedrosa, Katerina J. Argyraki, and George Candea. Verifying Software Network Functions with No Verification Expertise. In *ACM Symp. on Operating Systems Principles (SOSP)*, 2019.

[65] Minjia Zhang, Jipeng Huang, Man Cao, and Michael D Bond. Low-Overhead Software Transactional Memory with Progress Guarantees and Strong Semantics. In *Symp. on Principles and Practice of Parallel Computing (PPoPP)*, 2015.

[66] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps Intrusion Prevention on a Single Server. In *Symp. on Operating Systems Design and Implementation (OSDI)*, 2020.