State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing

Qiongwen Xu, *Rutgers University;* Sebastiano Miano, *Politecnico di Milano;* Xiangyu Gao and Tao Wang, *New York University;* Adithya Murugadass and Songyuan Zhang, *Rutgers University;* Anirudh Sivaraman, *New York University;* Gianni Antichi, *Queen Mary University of London and Politecnico di Milano;* Srinivas Narayana, *Rutgers University*

https://www.usenix.org/conference/nsdi25/presentation/xu-qiongwen

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

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

978-1-939133-46-5

Open access to the Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing

Qiongwen Xu[△], *Sebastiano Miano*[⊕], *Xiangyu Gao*[‡], *Tao Wang*[‡], *Adithya Murugadass*[△], *Songyuan Zhang*[△], *Anirudh Sivaraman*[‡], *Gianni Antichi* [Ω ⊕], *Srinivas Narayana*[△]

[△] *Rutgers University, USA* [⊕] *Politecnico di Milano, Italy* [‡] *New York University, USA* [Ω] *Queen Mary University of London, UK*

Abstract

With the slowdown of Moore's law, CPU-oriented packet processing in software will be significantly outpaced by emerging line speeds of network interface cards (NICs). Single-core packet-processing throughput has saturated.

We consider the problem of high-speed packet processing with multiple CPU cores. The key challenge is state—memory that multiple packets must read and update. The prevailing method to scale throughput with multiple cores involves state sharding, processing all packets that update the same state, e.g., flow, at the same core. However, given the skewed nature of realistic flow size distributions, this method is untenable, since total throughput is limited by single-core performance.

This paper introduces *state-compute replication*, a principle to scale the throughput of a single stateful flow across multiple cores using replication. Our design leverages a *packet history sequencer* running on a NIC or top-of-the-rack switch to enable multiple cores to update state without explicit synchronization. Our experiments with realistic data center and wide-area Internet traces show that state-compute replication can scale total packet-processing throughput linearly with cores, independent of flow size distributions, across a range of realistic packet-processing programs.

1 Introduction

Designing software to handle high packet-processing loads is crucial in networked systems. For example, software load balancers, CDN nodes, DDoS mitigators, and many other middleboxes depend on it. Yet, with the slowdown of Moore's law, software packet processing has struggled to keep up with line speeds of network interface cards (NICs), with emerging speeds of 200 Gbit/s and beyond [20]. Consequently, there have been significant efforts to speed up packet processing through better network stack design, removing user-kernel crossings, running software at lower layers of the stack (*e.g.,* NIC device driver), and designing better host interconnects.

We consider the problem of scaling software packet processing by using multiple cores on a server. The key challenge is that many packet-processing applications are *stateful*, maintaining and updating regions of memory across many packets. If multiple cores contend to access the same memory regions, there is significant memory contention and cache bouncing, resulting in poor performance. Hence, the classic approach to multicore scaling is to process packets touching distinct states, *i.e.,* flows, on different cores, hence removing memory contention and synchronization. For example, a load balancer that maintains a separate backend server for each 5-tuple may send all packets of a given 5-tuple to a fixed core, but process different 5-tuples on different cores, hence scaling performance with multiple cores. Many prior efforts have implemented and optimized such sharding-oriented solutions [35,46,54,63,68].

However, we believe that the existing approaches to multicore scaling have run their course (§2). Realistic traffic workloads have heavy-tailed flow size distributions and are highly skewed. With sharding, large "elephant flows" must be processed by one CPU core, reducing overall performance due to the low throughput of a single CPU core. With emerging 200 Gbit/s—1 Tbit/s NICs, a single packet processing core may be too slow to keep up even with a single elephant flow. Additionally, with the growing scales of volumetric resource exhaustion attacks, packet processors must gracefully handle attacks where adversaries force packets into a single flow [43].

This paper introduces a scaling principle, *state-compute replication (SCR)*, that improves the software packet-processing throughput for a *single, stateful flow* with additional cores, while avoiding shared memory and contention. Figure 1 shows how SCR scales the throughput of a single TCP connection for a TCP connection state tracker [40] with more cores, when other scaling techniques fail. A connection tracker may change its internal state with each packet.

SCR applies to any packet processing program that may be abstracted as a deterministic finite state machine. Intuitively, as long as each core can reconstruct the flow state by processing *all* the packets of the flow, multiple cores can successfully process a single stateful flow with zero cross-core synchronization. However, naively processing every packet with every CPU core cannot improve throughput with more

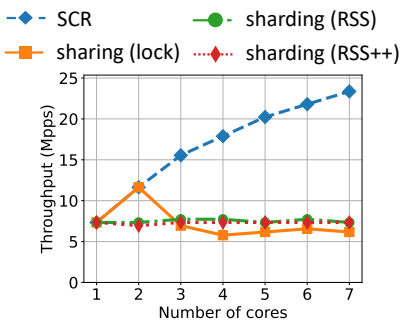


Figure 1: Scaling the throughput of a TCP connection state tracker for a *single TCP connection* across multiple cores. Sharing state across cores degrades performance beyond 2 cores due to contention. Sharding state (using RSS and RSS++ [35]) cannot improve throughput beyond a single CPU core (§2). In contrast, State-Compute Replication (§3) provides linear scale-up in throughput with cores.

cores. We need to meet two additional requirements. First, in systems where CPU usage is dominated by per-packet work, we must preserve the total number of packets moving through the system. Second, we must divide up *dispatch*—the CPU software labor of presenting the packets to the packet-processing program—across multiple cores (§3.1). This leads to the scaling principle below (more formal version in §3.1):
State-Compute Replication (informal). In a system bound by per-packet CPU work and software dispatch, *replicating* the state and the computation of a stateful flow across cores increases throughput *linearly* with cores, so long as we preserve the total number of packets traversing the system.

To meet the requirements for high performance in SCR, we design a *packet history sequencer* (§3.2), an entity which sees every packet and sprays packets across cores in a round-robin fashion, while piggybacking the relevant information from packets missed by a core on the next packet sent to that core. The sequencer must maintain a small bounded history of packet headers relevant to the packet-processing program. A high-speed packet-processing pipeline, running either on a programmable NIC or a top-of-the-rack switch, may serve as a sequencer. We present sequencer designs for two hardware platforms (§3.3), a Tofino switch, and a Verilog RTL module we synthesized into the NetFPGA-PLUS reference pipeline.

We evaluated the scaling efficacy of SCR using a suite of realistic programs and traffic (§4). SCR is the only technique we are aware of that monotonically scales the total processing throughput with additional cores regardless of the skewness in the arriving flow size distribution. However, no technique can continue scaling indefinitely. In §3.1 and §4, we demonstrate the limits of SCR scaling, along with the significant performance benefits to be enjoyed before hitting these limits. Additionally, we show the resource usage for our sequencer's RTL design, which meets timing at 340 MHz. We believe this design is simple and cheap enough to be added as an on-chip accelerator to a future NIC.

All our software is publicly available [31]. This work does not raise any ethical concerns.

2 Background and Motivation

2.1 High Speed Packet Processing

This paper considers packet-processing programs that must work at heavy network loads with quick turnarounds for packets. We are specifically interested in applications implementing a "hairpin" traffic flow: receiving a packet, processing it as quickly as possible, and transmitting the packet right out, at the highest possible rate. Such programs exhibit compute and working set sizes that are smaller in comparison to full-fledged applications (running in user space) or transport protocols (TCP stacks) running at endpoints.

Examples of the kinds of applications we consider include (i) middlebox workloads (network functions), such as firewalls, connection trackers, and intrusion detection/prevention systems; and (ii) high-volume compute-light applications such as key-value stores, telemetry systems, and stream-based logging systems, which process many requests with a small amount of computation per request [51]. A key characteristic of such applications is their need for high packet-processing rate (which is more important than byte-processing rate) and the fact that their performance is primarily bottlenecked by CPU usage [44, 69, 73].

The performance of such applications is mission-critical in many production systems. Even small performance improvements matter in Internet services with large server fleets and heavy traffic. As a specific example, Meta's Katran layer-4 load balancer [8] and CloudFlare's DDoS protection solution [44] process every packet sent to those respective services, and must withstand not only the high rate of request traffic destined to those services but also large denial-of-service attacks. More generally, the academic community has undertaken significant efforts for performance optimization of network functions, including optimization of the software frameworks [46], designing language-based high-performance isolation [61], and developing custom hardware offload solutions [64].

In this paper, we consider applications developed within high-speed packet processing software frameworks. Given the slowdown of Moore's law and the end of Dennard scaling, the software packet-processing performance of single CPU cores has saturated. Even expert developers must work meticulously hard to improve per-core throughput by small margins (like 10%) [37, 50, 52, 67]. The community has pursued various efforts to improve performance, such as re-architecting the software stack to make efficiency gains [21, 56, 69], introducing stateless hardware offloads working in conjunction with the software stack [19, 29], full-stack hardware offloads [30, 34], and kernel extensions [2]. This paper considers software frameworks that modify the device driver to enable programmable and custom functionality to be incorporated by developers at high performance with minimal intervention from the existing kernel software stack [50].

Specifically, we study performance and evaluate our tech-

niques in the context of kernel extensions implemented using the eXpress Data Path (XDP/eBPF [50]) within the Linux kernel. In §3, we will discuss how our observations and principles apply more generally to other high-speed software packet-processing frameworks, including those written with user-space libraries like DPDK [21].

2.2 Parallelizing Stateful Packet Processing

This paper considers the problem of parallelizing high-speed packet processing programs across multiple cores. The key challenge is handling *state*: memory that must be updated in a sequential order upon processing each packet to produce a correct result. Consider the example of the connection tracker [40], a program which identifies the TCP connection state (*e.g.,* SYN sent, SYN/ACK sent, *etc.*) using packets observed from both directions of a TCP connection. Each packet in the connection may modify the internal connection state maintained by the program. There are two main techniques used to parallelize such programs across cores.

Shared state parallelism. One could conceive a parallel implementation that (arbitrarily) splits packets across multiple cores, with explicit synchronization or retry logic guarding access to the shared memory, *i.e.,* the TCP connection state, to ensure a correct result.

Shared-state parallelism works well when the contention to shared memory is low. Specifically, shared-memory scaling could work well when (i) packets of a single flow arrive slowly enough, *e.g.,* if there are a large number of connections with a roughly-equal packet arrival rate, or (ii) when there are efficient implementations available for synchronization or atomic updates in software [47] or hardware [18, 25]. However, neither of these conditions are generally applicable. Many flow size distributions encountered in practical networks are highly skewed [36, 75] or exhibit highly bursty behavior [70], resulting in significant memory contention if packets from the heavier flows are spread across cores. Further, the state update operation in many programs, including the TCP connection tracker, are too complex to be implemented directly on atomic hardware, since the latter only supports individual arithmetic and logic operations (like fetch-add-write). Our evaluation results (§4) show that the performance of shared-state multicore processing plummets with more cores under realistic flow size distributions.

Sharded (shared-nothing) parallelism. Today, the predominant technique to scale stateful packet processing across multiple cores is to process packets that update the same memory at the same core, sharding the overall state of the program across cores. Sharding is achieved through techniques like Receive Side Scaling (RSS [4]), available on modern NICs, to direct packets from the same flow to the same core, and using shared-nothing data structures on each core.

However, sharding suffers from a few disadvantages. First, it is not always possible to avoid coordination through sharding. There may be parts of the program state that are shared across all packets, such as a list of free external ports in a Network Address Translation (NAT) application. On the practical side, the RSS implementations on today's NICs partition packets across cores using a limited number of combinations of packet header fields. For example, a NIC may be configured to steer packets with a fixed combination of source and destination IP addresses (but not a fixed source IP address) to a fixed CPU core. Further, the granularity at which the application wants to shard its state—for example, a key-value cache may seek to shard state by the key requested in the payload—could be infeasible to implement with the packet header sets supported by the RSS capabilities of the NIC [63].

Second, sharding state may create load imbalance across cores if some flows are heavier or more bursty than others, creating hotspots on single CPU cores. Skewed flow size distributions [36], bursty flow transmission patterns [70], and denial of service attacks [43] create conditions ripe for such imbalance. The research community has investigated solutions to balance the packet processing load by migrating flow shards across CPU cores [35, 63]. However, the efficacy of re-balancing is limited by the granularity at which flows can be migrated across cores. As we show in our evaluation (§4), the throughput of the heaviest and most bursty flows is still limited by a single CPU core, which in turn limits the total achieved throughput. Another alternative is to evenly spray incoming packets across cores [41, 71], assuming that only a small number of packets in each flow need to update the flow's state. If a core receives a packet that must update the flow state, the packet is redirected to a designated core that performs all writes to the state for that flow. However, the assumption that state is mostly read-only is not universal, *e.g.,* a TCP connection tracker may update state potentially on every packet. Further, packet reordering at the designated core can lead to incorrect results [35].

2.3 Goals

Given the drawbacks of existing approaches for multi-core scaling discussed above (§2.2), we seek a scaling technique that achieves the following goals:

1. *Generic stateful programming.* The technique must produce correct results for general stateful updates, eschewing solutions that only work for "simple" updates (*i.e.,* fitting hardware atomics) or only update state for a small number of packets per flow.
2. *Skew independence.* The scaling technique should improve performance independent of the incoming flow size distribution or how the flows access the state in the program.
3. *Monotonic performance gain.* Performance should improve, not degrade or collapse, with additional cores.

3 State-Compute Replication (SCR)

In §3.1, we present scaling principles for multi-core stateful packet processing, to meet the goals in §2.3. In §3.2 through §3.4, we show how to operationalize these principles.

3.1 Scaling Principles

To simplify the discussion, suppose the packet-processing program is *deterministic*, *i.e.,* in every execution, it produces the same output state and packet given a fixed input state and packet (we relax this assumption in §3.4).

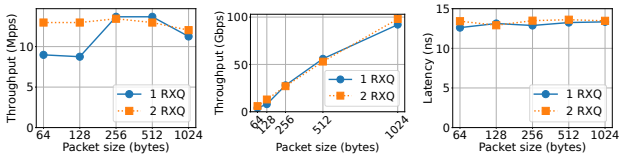
Principle #1 (Replication for correctness). Sending every packet reliably to every core, and *replicating* the state and computation on every core, produces the correct output state and packet on every core *with no explicit cross-core synchronization,* regardless of how the state is accessed by packets.

This principle asks us to treat each core as one replica of a replicated state machine running the deterministic packet-processing program. Each core processes packets in the same order, without missing any packets. With each incoming packet, each core updates a private copy of its state, which is equal to the private copy on every other core. There is no need to synchronize explicitly. Further, replication provides the benefit that the workload across cores is even regardless of how the state is accessed by packets, *i.e.,* skew-independent.

One way to apply this principle naively is to broadcast every packet received *externally* on the machine to every core: with k cores, for each external packet, the system will process k *internal* packets, due to k-fold packet duplication. However, artificially increasing the number of packets processed by the system will significantly hurt performance. In CPU-bound packet processing, smaller packets typically require the same computation that larger packets do. That is, the total amount of work performed by the system is proportional to the packets-per-second offered, rather than the bits-per-second [50, 69].

So how should one use replication for multi-core scaling? Understanding the dominant components of the per-packet CPU work in high-speed packet-processing frameworks offers insight. There are two parts to the CPU processing for each packet after the packet reaches the core where it will be ultimately processed: (i) *dispatch*, the CPU/software labor of presenting the packet to the user-developed packet-processing program, and signaling the packet(s) emitted by the program for transmission by the NIC; and (ii) the *program computation* running within the user-developed program itself. Dispatch often dominates the per-packet CPU work [50].

While these observations are known in the context of high-speed packet processing, we also benchmarked a simple application on our own test machine to validate them. Consider Figure 2, where we show the throughput (packets/second (a), bits/second (b)), and latency (c) of a simple packet forwarder written in the XDP framework running on a single CPU core. Our testbed setup is described in much more detail later (§4),



(a) Packets/second (b) Bits/second (c) Latency (ns)

Figure 2: The nature of CPU work in high-speed packet processing: Consider the throughput of a simple packet forwarding application (packets/second (a), bits/second (b)) running on a single CPU core clocked at 3.6 GHz, as the size of the incoming packets varies. The average latency to execute the XDP program is also shown in nanoseconds (c). CPU usage is tied to the number of packets (not bits) processed per second. Further, significant time elapses in *dispatch:* CPU work to present the input packet to and retrieve the output packet from the program computation.

but we briefly note here that our device-under-test is an Intel Ice Lake CPU configured to run at a fixed frequency of 3.6 GHz and attached to a 100 Gbit/s NIC. At each packet size below 1024 bytes, the CPU core is fully utilized (at 1024 bytes the bottleneck is the NIC bandwidth). The achieved packets/second is stable across all packet sizes which are CPU bound (see the *2 RXQ* curve). With a processing latency of roughly 14 nanoseconds at all packet sizes (measured only for the XDP forwarding program), back-to-back packet processing should "ideally" process $(14 * 10^{-9})^{-1} \approx 71$ million packets/second. However, the achieved best-case throughput ($\approx$ 14 million packets/second) is much smaller—implying that significant CPU work is expended in presenting the input packets to and extracting the output packets from the forwarder and setting up the NIC to transmit and receive those packets. This is not merely a feature of the framework we used (XDP); the DPDK framework has similar dispatch characteristics [50].

Our key insight is that it is possible to replicate program computation without replicating dispatch, enabling multi-core performance scaling. This leads us to the next principle:

Principle #2 (State-Compute Replication). Piggybacking a *bounded recent packet history* on each packet sent to a core allows us to use replication (#1) while equalizing the external and internal packets-per-second CPU work in the system.

Principle #2 states that replication (principle #1) is possible without increasing the total internal packet rate or per-packet CPU work done by the system. Suppose it is possible to spray the incoming packets across cores in a round-robin fashion. If there are k cores, each core receives every k^{th} packet. Then:

1. It is unnecessary for each core to have the most up-to-the-packet private state at all times. For correctness, it is sufficient if a core has a state that is "fresh enough" to make a decision on the current packet that it is processing.
2. With each new packet, suppose the core that receives it also sees (as metadata on that packet) all the $k-1$ packets from the last time it processed a packet, *i.e., a recent packet history*. The core can simply "catch up" on the computations required to obtain the most up-to-the-packet value

for its private state.

3. If packets are sprayed round-robin across cores, the number of historic packets needed to ensure that the most updated state is available to process the current packet is equal to the number of cores. Further, just those packet bits required to update the shared state are necessary from each historic packet, allowing us to pack multiple packets' worth of history into a single packet received at a core.

As a simple model, suppose a system has k cores, and each core can dispatch a single packet in d cycles and runs a packet-processing program that computes over a single packet in c cycles. For each piggybacked packet, the total processing time is $d + (k \times c)$. When dispatch time dominates compute time ($d \gg c$), with k cores, the total rate at which externally-arriving packets can be processed is $k \times \frac{1}{d+(k\times c)} \approx k/d$. Hence, it is possible to scale the packet-processing rate linearly with the number of cores k. In Appendix A, we show that a model like this indeed accurately predicts the empirical throughput achieved with a given number of cores.

Intuitively, doing some extra "lightweight" program computation per packet enables scaling the "heavyweight" dispatch computation with more cores, while maintaining correctness.

Principle #3 (Scaling limits). Principle #2 provides a linear scale-up in the packets-per-second throughput with more cores, so long as dispatch dominates the per-packet work.

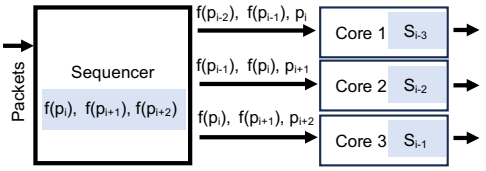
The scaling benefits of principle #2 taper off beyond a point. Dispatch can be overtaken as the primary contributor to per-packet CPU work, for example, when (i) the compute time $k \times c$ for each piggybacked packet becomes sizable; (ii) the per-packet compute time c itself increases due to overheads in the system, *e.g.,* larger memory access time when a core's copy of the state spills into a larger memory; or (iii) other components such as the NIC or PCIe become the bottleneck rather than the CPU. When this happens, the approximation in our simple linear model ($d \gg c$) no longer holds, and the system's packet rate no longer scales linearly with cores.

3.2 Operationalizing SCR

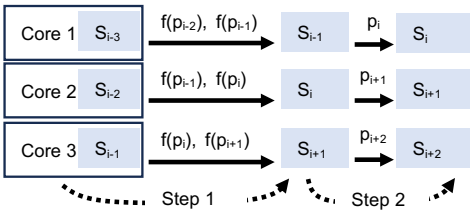
Operationalizing the scaling principles discussed above (§3.1) conceptually requires two pieces.

A reliable packet history sequencer (§3.3). We require an additional entity in the system, which we call a *sequencer,* to (i) steer packets across cores in round-robin fashion, (ii) maintain the most recent packet history across all packets arriving at the machine, and (iii) piggyback the history on each packet sent to the cores. After the packet is processed by a CPU core, its piggybacked history can be stripped off on the return path either at the core itself or at the sequencer. The size of the packet history depends only on the number of cores and metadata size (§3.1) and is independent of the number of active flows.

The NIC hardware or the top-of-the-rack switch are natural points to introduce the sequencer functionality, since they observe all the packets flowing into and out of the machine. Today's existing fixed-function NICs do not implement the functionality necessary to construct and piggyback a reliable packet history. However, we have identified two possible instantiations that could, in the near future, achieve this: (i) emerging NICs, *e.g.,* with programmable pipelines [1, 3, 27], implementing the full functionality of the reliable sequencer; or (ii) a combination of a NIC implementing round-robin packet steering [4, 7] and a programmable top-of-the-rack switch pipeline [17, 23, 38] for maintaining and piggybacking the packet history. We believe that either of these instantiations may be realistic and applicable given the context: for example, high-speed programmable NICs are already common in some large production systems [45], as are programmable switch pipelines [60]. We will show two possible hardware designs in §3.3. Hereafter, for brevity, we refer to both of these designs as simply sequencers.



(a) The sequencer stores relevant fields from the packet history, and piggybacks the history on packets sprayed round-robin across cores.



(b) Each core fast-forwards its private state and then handles its packet.

Figure 3: An example illustrating the scaling principles. p_i is the i^{th} packet received by the sequencer, $f(p_j)$ are relevant fields from p_j, and S_i is the state after processing packets $p_1, \ldots, p_i$ in order.

An SCR-aware packet-processing program. The packet-processing program must be developed to replicate the program state and keep private copies per core. Further, the program must process the packet history first before computing over the current packet. In Appendix C, we show an example that demonstrates how to transform a single-threaded program to its SCR-aware variant. We believe that these program transformations can be automated in the future.

An example showing scaling principles in action. Consider Figure 3, where a sequencer and three cores are used to run a packet-processing program. As shown in Figure 3a, the sequencer sprays packets (*i.e.,* $p_i, p_{i+1}, \ldots$) in a round-robin fashion across $k = 3$ cores (*i.e.,* $core_1, core_2, core_3$). Further, the sequencer stores the recent packet history consisting of the packet fields from the last k packets which are relevant to evolving the flow state. We denote the relevant part of a

packet p_i by $f(p_i)$. For example, in a TCP connection tracking program, this includes the TCP 4-tuple, the TCP flags, and sequence and ACK numbers. Note that this packet history is updated only by the sequencer and is never written to by the cores. In the example in Figure 3a, the packet history supplied to $core_1$ processing packet p_i is $f(p_{i-2}), f(p_{i-1})$. As shown in Figure 3b, each core updates its local private state, first fast-forwarding the state by running the program through the packet history $f(p_{i-2}), f(p_{i-1})$, and then processing the packet p_i sprayed to it.

If the packet-processing program is deterministic (§3.1), an SCR-aware program is guaranteed to produce the correct output state and packet if every CPU core is guaranteed to (losslessly) receive the packets sent to it by the sequencer. We show how to handle non-determinism and packet loss in §3.4.

3.3 Packet History Sequencer

The primary goal of the sequencer is to maintain and propagate recent packet history to CPU cores to help replicate the computation with the correct program state (§3.2). We assume the NIC is already capable of spraying packets across CPU cores [4, 7], and hence do not discuss that functionality further. We describe the rest of the sequencer's functions in terms of the following: (i) designing a packet format that modifies existing packets to piggyback history from the sequencer to the CPU cores; (ii) designing a hardware data structure that maintains a recent bounded packet history at the sequencer, and enables reading out the history into metadata on the packet. The packet fields that are maintained in the sequencer history depend on the specific fields used by the packet-processing application. The number of historic packets that must be tracked depends on the degree of parallelism that is sought, *e.g.,* the number of available CPU cores over which scaling is implemented.

We have implemented sequencing hardware data structures on two platforms, the Tofino programmable switch pipeline [23] and a Verilog module that we integrated into the NetFPGA-PLUS project [12].

3.3.1 Packet format

The key question answered in this subsection is: given a packet, what is the best place to put the packet history on it? While this may initially appear to be just an engineering detail, the packet format has important implications to the design of hardware data structures on the sequencer and the SCR-aware program.

As shown in Figure 4a, we choose to place the packet history close to the beginning of the packet, before the entirety of the original packet. Relative to placing the packet history between headers of the original packet, this placement simplifies the hardware logic that writes the history into the packet, as the write must always occur at a fixed address (0) in the packet buffer. Further, for reasons explained in §3.3.2, we include a pointer to the metadata of the packet that arrived the earliest among the ones in the piggybacked history. The earliest packet does not always correspond to the first piece of metadata when reading the bytes of the packet in order.

Keeping all the bytes of the original packet together in one place also simplifies developing an SCR-aware packet-processing program. The packet parsing logic of the original program can remain unmodified if the program starts parsing from the location in the modified packet buffer which contains all the bytes of the original packet in order.

Finally, we also prefix an additional Ethernet header to the packet in instantiations of the sequencer which run outside of the NIC, *i.e.,* a top-of-the-rack switch. Adding this header helps the NIC process the packet correctly: without it, the packet appears to have an ill-formatted Ethernet MAC header at the NIC. Our setup also uses this Ethernet header to force RSS on the NIC [4] to spray packets across CPU cores (our testbed NIC (§4.1) supports hashing on L2 headers). This additional Ethernet header is not needed in a sequencer instantiation running on the NIC.

3.3.2 Hardware data structures for packet history

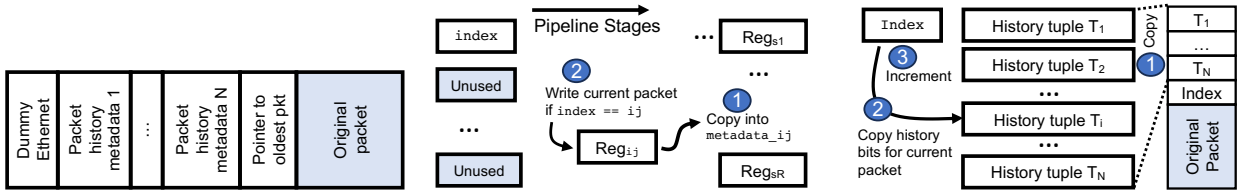
We show how to design data structures to maintain and update a recent packet history on two high-speed platforms, a Tofino programmable switch pipeline [23] and a Verilog module integrated into the NetFPGA-PLUS platform [12]. These designs are specific to the platform where they are implemented, and hence we describe them independently.

A key unifying principle between the two designs is that although the items in the maintained packet history change after each packet, we only update a small part of the data structure for each packet. Conceptually, a ring buffer data structure is appropriate to maintain such histories. Hence, in both designs, we use an *index pointer* to refer to the current data item that must be updated, which corresponds to the head pointer of the abstract ring buffer where data is written.

Tofino. We use Tofino registers [26], which are stateful memories to hold data on the switch, to record the bits of each historic packet relevant to the computation in the packet-processing program. We use the Tofino parser to extract these bits, which restricts our design only to support historic fields up to 4 Kilobits deep in the packet [38].

Suppose the pipeline has s match-action table stages, R registers per stage, and b bits per register. For simplicity in this description, we assume there is exactly one packet field of size b bits used in the computation in the packet-processing program. Our data structure can maintain a maximum of $(s-1) \times R \times b$ bits of recent packet history, *i.e.,* history for $(s-1) \times R$ packets, as shown in Figure 4b. We have successfully compiled the design to the Tofino ASIC.

First, we use a single register in the first stage to store the index pointer. The pointer refers to the specific register in the



(a) Packet format (b) Tofino sequencer (c) RTL sequencer

Figure 4: Hardware data structures. (a) Packets modified to propagate history from the sequencer to CPU cores. The sequencer prefixes the packet history to the original packet, which allows for a simpler implementation in hardware (§3.3) and simpler transformations to make a packet-processing program SCR-aware (Appendix C). In instantiations where the sequencer is partly implemented on a top-of-the-rack switch (§3.2), we further prefix a dummy Ethernet header to ensure that the NIC can process the packet correctly. (b) The data structure used to maintain and propagate packet history on the Tofino programmable switch pipeline (§3.3.2). Inset shows the specific actions performed on each Tofino register. (c) The data structure used to maintain and propagate packet history in our Verilog module on NetFPGA-PLUS (§3.3.2).

subsequent stages that must be updated with a header field from the current packet. The index pointer is incremented by 1 for each packet, and wraps back to 0 when it reaches the maximum number of packets required in the history. The pointer is also carried on a metadata field on the packet through the remaining pipeline stages.

Next, register ALUs in subsequent stages are programmed to read out the values stored in them into pre-designated metadata fields on the packet. If the index pointer points to this register, an additional action occurs: rewrite the stored contents of the register by the pre-designated history fields from the current packet.

Finally, all the metadata fields, consisting of the packet history fields and the index pointer, are deparsed and serialized into the packet in the format shown in Figure 4a. We also add an additional Ethernet header to ensure that the server NIC can receive the packets correctly (§3.3.1).

Recent work explored the design of ring buffers to store packet histories in the context of debugging [53], reading out the histories from the control plane when a debugging action is triggered. A key difference in our design is that reading out histories into the packet is a data plane operation, occurring on every packet.

NetFPGA. To show the possibility of developing high-speed fixed-function hardware for sequencing, we also present a sequencer design developed in Verilog in Figure 4c.

Suppose we wish to maintain a history of N packets, each packet contributing b bits of information. A simple design, for small values of N and b (we used $N = 16$ and $b = 112$), uses a memory which has N rows, each containing a tuple of b bits. We also maintain a register containing the index pointer (p bits), initialized to zero. At the beginning, the memory is initialized with all zeroes. When a packet arrives, it is parsed to extract the bits relevant to the packet history. Then the entire memory is read and put in front of the packet (moving the packet contents by a fixed size known beforehand, $N \times b + p$ bits). The information relevant to the packet history from the current packet is put into the memory row pointed to by the index pointer, and the index pointer is incremented (modulo the memory size). We have integrated this design into the NetFPGA-PLUS platform.

3.4 Discussion

In this section, we discuss how to generalize SCR to handle non-determinism, packet loss, and multiple packet-processing programs.

Handling Packet Loss. Packets can be lost either prior to the sequencer, after the sequencer but prior to processing at a CPU core, or after processing at a core. Among these, we only care about the second kind of packet loss, since this is the only one that is problematic specifically for SCR, introducing the possibility that flow states on the CPU cores might become inconsistent with each other.

We expect that SCR will be deployed in scenarios where the event of packet loss between the sequencer and CPU cores is rare. First, we do not anticipate any packet loss in an instantiation where the sequencer is running entirely on a NIC, since the host interconnect between the NIC and CPU cores uses credit-based flow control and is lossless by design [32, 59]. In an instantiation where the sequencer is running on a top-of-the-rack switch, it is possible to run link-level flow control mechanisms like PFC [6] (as some large production networks do [48]) to prevent packet loss between the switch and server cores. We discuss below how SCR can handle rare packet drop and corruption events while maintaining consistency among the states on CPU cores.

How should a CPU core that has lost a packet arriving from the sequencer synchronize itself to the correct flow state? There are two design options: the core can either explicitly read the full flow state from a more up-to-date core, or it can read the packet history from either the sequencer or a log written by a more up-to-date core, and then use the history to catch up its private state (akin to Figure 3). Since we operate in a regime where packet losses are rare, but the full set of flow states is large, we prefer to synchronize the packet history rather than the state. Further, to simplify the overall design, we avoid explicit coordination between the cores and the sequencer, synchronizing the history among the cores only.

Our objective is *atomicity:* any packet is either processed by all the cores or none of the cores. If a packet is sent by the sequencer to *any* core (in original or as part of the packet history), it should be processed in the correct order by *all* the cores. To achieve this, we (i) have the sequencer attach an incrementing sequence number to each packet released by it; (ii) use a per-core, lockless, single-writer multiple-reader log, into which each core writes the history contained in each packet it receives (including the relevant data for the original packet); and (iii) introduce an algorithm to catch up the flow state on each core upon detection of loss.

The algorithm proceeds as follows (more information is available in Algorithm 1 in Appendix B). Each CPU core c maintains a per-core log with one entry for each sequence number i. In a system with N cores, the history metadata of the packet with sequence i (say $history[i]$) will appear in packets with sequence numbers i through $i+N-1$; conversely, a packet with sequence number j contains $history[minseq], \cdots, history[j]$ where $minseq \triangleq max(1, j-N+1)$.

For core c and sequence number i, $log[c][i]$ takes on one of three possible values:

$$\begin{cases} history[i] & \text{if history for sequence } i \text{ was received at } c \\ NOT\_INIT & \text{if the highest sequence received at } c \text{ is } j < i \\ LOST & \text{if } c \text{ has received sequence } j > i, \\ & \text{but sequence } i \text{ was not received at } c \end{cases}$$

At the beginning, for all cores c and sequence numbers i, $log[c][i]$ is set to NOT_INIT, to denote that each log entry is uninitialized at every core. When a (fixed) core c receives a packet with sequence number j, it first detects packet loss by comparing the max sequence number it has seen so far (say $max[c]$) with the earliest sequence number in the new packet it receives ($minseq$), assuming no reordering between the sequencer and the core. Then, c processes every sequence number k such that $max[c] < k \leq j$, in order of increasing k, as follows:

1. if $k < minseq$, *i.e.,* sequence k was lost between the sequencer and core c, the core updates $log[c][k] \leftarrow LOST$. For such packets, core c will read from the logs of other cores $c' \neq c$ in a loop, until c discovers either that (i) $history[k]$ is written in $log[c'][k]$, in which case c catches up its private state by reading this history; or (ii) $log[c'][k] = LOST$ on all cores $c' \neq c$, concluding that sequence k was never originally received on any core, and does not need to be recovered for atomicity;
2. if $minseq \leq k \leq j$, *i.e.,* sequence k is successfully received at core c (as part of the current packet), core c updates $log[c][k] \leftarrow history[k]$ available in the packet, and then proceeds with regular processing as in §3.2.

In Appendix B, we formally prove that, under some mild assumptions, this algorithm always terminates in a state that is eventually consistent across all CPU cores. Despite cores possibly waiting on one another, there will be no deadlocks. While the treatment above uses an infinitely large log, practically, our log implementation is a circular buffer with a fixed size (it is unnecessary to garbage-collect the log). The buffer must be sized large enough to recover from packet losses and transient speed mismatches across CPU cores, both of which are expected to be small in practical deployments.

Handling programs that depend on timestamps. The use of timestamps measured locally at each CPU core (*e.g.,* to implement a token bucket rate limiter) may result in the results of computations at different CPU cores diverging from each other. To handle this, we avoid measuring time locally at each CPU core, and instead have the sequencer attach a timestamp for each packet to the packet history. Modern NICs and programmable switches support high-resolution hardware timestamping over packets [13, 26].

Handling programs involving randomization. For SCR to produce a consistent state across cores, it is necessary that the state computations on all CPU cores agree on the result even if the computations involve random numbers. None of our current benchmark programs involve randomization. For those programs that do, we recommend to fix the seed of the pseudorandom number generator to the same value across different CPU cores.

Handling chained packet-processing programs. SCR can handle multiple packet-processing programs run sequentially (for example, for service function chaining [49]) by piggybacking the union of the historical packet fields for all the programs on each packet from the sequencer to the core. In addition to the program changes typically needed for SCR parallelism (Appendix C), the programs must also be rewritten to handle packets that include additional fields of history for the co-resident programs. We believe this can be accomplished by the design of a suitable automatic compiler. We leave the design of such compilers to future work.

4 Evaluation

We seek to answer two main questions through the experiment setup described in §4.1.
(1) Does state-compute replication provide better multi-core scaling than existing techniques (§4.2)?
(2) How practical is sequencer hardware (§4.3)?

4.1 Experiment Setup

Machines and configurations. Our experiment setup consists of two server machines connected back-to-back over a 100 Gbit/s Nvidia/Mellanox ConnectX-5 NIC on each machine. Our servers run Intel Ice Lake processors (Xeon Gold 6334) with 16 physical cores (32 hyperthreads) and 256 GB DDR4 physical memory spread over two NUMA nodes. The system bus is PCIe 4.0 16x. We run Ubuntu 22 with Linux kernel v6.5.

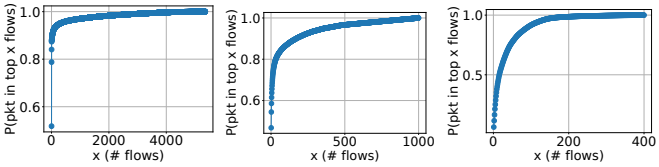
Program	State		Metadata size	RSS hash	Packet traces	Atomic HW	Lines of code
	Key	Value	(bytes/packet)	fields	evaluated	vs. Locks	(shard/RSS)
DDoS mitigator	source IP	count	4	src & dst IP	CAIDA, Univ DC	Atomic HW	168
Heavy hitter monitor	5-tuple	flow size	18	5-tuple	CAIDA, Univ DC	Atomic HW	141
TCP connection state tracking	5-tuple	TCP state, timestamp, seq #	30	5-tuple	Hyperscalar DC	Locks	1029
Token bucket policer	5-tuple	last packet timestamp, # tokens	18	5-tuple	CAIDA, UnivDC	Locks	169
Port-knocking firewall	source IP	knocking state (*e.g.,* `OPEN`)	8	src & dst IP	CAIDA, UnivDC	Locks	123

Table 1: The packet-processing programs we evaluated.

One machine serves as a packet replayer/generator, running a DPDK burst-replay program which can transmit packets from a traffic trace. We have tested that the traffic generator can replay large traces (1 million packets) at speeds of $\sim$ 120 million packets/second (Mpps), for sufficiently small packets (so that the NIC bandwidth is not saturated first). The traffic generator can be directed to transmit packets at a fixed transmission (TX) rate and measure the corresponding received (RX) packet rate. Our second server is the Device Under Test (DUT), which runs on identical hardware and operating system as the first server. We implement standard configurations to benchmark high-speed packet processing [50]: hyperthreading is disabled; the processor C-states, DVFS, and TurboBoost are disabled; dynamic IRQ balancing is disabled; and the clock frequency is set to a fixed 3.6 GHz. We enable PCIe descriptor compression and use 256 PCIe descriptors. Receive-side scaling (RSS [4]) is configured according to the baselines/programs, see Table 1. We use a single receive queue (RXQ) per core unless specified otherwise. On our setup, we have checked that the per-core packet-forwarding throughput is comparable to prior work benchmarking the XDP framework [50]. However, the absolute numbers are lower than the multi-core throughput reported in recent literature using DPDK. We believe this may be due to the differences in the packet-processing framework (XDP vs. DPDK).

The definition of throughput. We use the standard *maximum loss-free forwarding rate* (MLFFR [5]) methodology to benchmark packet-processing throughput. Our threshold for packet loss is in fact larger than zero (we count $< 4\%$ loss as "loss-free"), since, at high speeds we have observed that the software typically always incurs a small amount of bursty packet loss. We use binary search to expedite the search for the MLFFR, stopping the search when the bounds of the search interval are separated by less than 0.4 Mpps. Experimentally, we observe that MLFFR is a stable throughput metric: we get highly repeatable results across multiple runs. We only report throughput from a single run of the MLFFR binary search.

Traces. We are interested in understanding whether SCR provides better multi-core scaling than existing techniques on realistic traffic workloads. We have set up and used three traces for throughput comparison: a university data center trace [36], a wide-area Internet backbone trace from CAIDA [11], and a synthetic trace with flows whose sizes and inter-arrivals were sampled from a hyperscalar's data center flow characteristics [33]. These traces are highly dynamic, with flow states being created and destroyed throughout—an aspect that we believe is crucial to handle in real deployment environments (*i.e.,* the programs are not simply processing a stable set of active flows). Further, we ensure that all TCP flows that begin in the trace also end, by setting TCP SYN and FIN flags for the first and last packets (respectively) of each flow in the trace. This allows the trace to be replayed multiple times with the correct program semantics. The flow size distributions of the traces are shown in Figure 5.



(a) University DC (b) Internet backbone (c) Hyperscalar DC

Figure 5: Flow size distributions of the packet traces we used. We used real packet traces captured at (a) university data center [36] and (b) wide-area Internet backbone by CAIDA [11]. We also synthesized (c) a packet trace with real TCP flows whose sizes are drawn from Microsoft's data center flow size distribution [33].

The eBPF framework limits our implementations in terms of the number of concurrent flows that our stateful data structures can include. This is not a limitation of the techniques, but an artifact of the current packet-processing framework we use (eBPF/XDP). To account for this limitation, specifically for the CAIDA trace, we have sampled flows from the trace's empirical flow size distribution to faithfully reflect the underlying distribution, without over-running the limit on the number of concurrent flows that any of our baseline programs may hold across the lifetime of the experiment.

Baselines. We compare state-compute replication against (i) state sharing, an approach that uses either hardware atomic instructions (when the stateful update is simple enough) or eBPF spinlocks [10] (when it is not) to share state across CPU cores; (ii) state sharding using classic RSS; and (iii) sharding using RSS++ [35, 63], the state-of-the-art flow sharding technique to balance CPU load. RSS++ solves an optimization problem that takes as input the incoming load imposed by flow shards, and migrates shards to minimize a linear combination of load imbalance across CPU cores and the number of cross-core shard transfers needed. Both SCR and state sharing spray packets evenly across CPU cores. The packets sent to

each core for the sharding techniques depends on the configuration of RSS, which varies across the programs we evaluated (see below and Table 1). Running RSS++ over eBPF/XDP requires patching the NIC driver [9] to expose the RX hash on packets to XDP programs. Unless specified otherwise, we run SCR without loss recovery (§3.4) as we believe this is the most representative scenario in which SCR will be deployed. We evaluate loss recovery separately (§4.2).

Programs. We tested five packet-processing programs developed in eBPF/XDP, including (i) a heavy hitter monitor, (ii) DDoS mitigator, (iii) TCP connection state tracker, (iv) port-knocking firewall, and (v) a token bucket policer. Table 1 summarizes these programs. Each program maintains state across packets in the form of a key-value dictionary, whose size and contents are listed in the table. We developed a cuckoo hash table to implement the functionality of this dictionary with a single BPF helper call [15]. The packet fields in the key determine how RSS must be configured: packets having the same key fields must be sent to the same CPU core. However, today's NICs do not allow RSS to steer packets on arbitrary sets of packet fields [63]. We pre-process the trace to ensure that RSS hashing indeed shards the flow state correctly. For example, on the NIC in our testbed, the source (`srcip`) and destination (`dstip`) IP addresses may be used together, but not separately, as the RSS key to hash a packet to a core. For a program that maintains flow state at the granularity of `dstip`, an RSS key (`srcip, dstip`) could steer two packets with the same `dstip` but different `srcip` to different CPU cores, violating sharding at the granularity of `dstip`. To prevent this, and to evaluate our sharding baselines fairly, we pre-process our traces (*e.g.,* modifying packets such that every `srcip, dstip` combination in the trace hashes to a core that only depends on `dstip`) to ensure that RSS hashing indeed shards the flow state correctly. For the connection tracker, since both directions of the connection must go to the same CPU core, we use the keyed hash function prescribed by symmetric RSS [74].

4.2 Multi-Core Throughput Scaling

In this section, we compare the MLFFR throughput (§4.1) of several packet-processing programs (Table 1) scaled across multiple cores using four techniques: SCR (§3), state sharing with packets sprayed evenly across all cores, sharding using RSS, and sharing using RSS++ (§2). Since the TCP connection tracking program requires packets from the two directions of the connection to be aligned, we evaluated it on a synthetic but realistic hyperscalar data center trace (§4.1). For the rest of the programs, we report results from real university data center and Internet backbone traces.

We have ensured that these experiments reflect a fair comparison of CPU packet-processing efficacy. First, we truncated the packets in the traces to a size smaller than the full MTU, to stress CPU performance with a high packets/second (Mpps) workload (§3.1). Further, we fix the packet sizes used across all baselines for a given program. We used a fixed packet size of 256 bytes for the connection tracker and 192 bytes for the others. The packet size limits the number of items of history metadata that can be piggybacked on each packet. Since the metadata size changes by the program (Table 1), the maximum number of cores we can support for a fixed packet size also varies by program (we support 7 cores for the token bucket, heavy hitter detector, and connection tracker, and 14 for the DDoS mitigation and port-knocking firewall).

Throughput results. Figure 6 and Figure 7 show the throughput as we increase the number of packet-processing cores. SCR is the only multi-core scaling technique that can monotonically scale the throughput of all the stateful packet-processing programs we evaluated across multiple cores, regardless of the flow size distribution (§2.3). The throughput for SCR increases linearly across cores in all of the configurations we tested. Somewhat surprisingly, SCR provides even better absolute performance than hardware atomic instructions in the case of the heavy hitter and DDoS mitigation programs. However, the performance of lock-based sharing falls off catastrophically with 3 or more cores.

The throughput of sharding using RSS depends on how the RSS hash function steers flows to cores. On our experimental setup, RSS can split flows evenly across CPU cores. However, RSS neither splits individual heavy flows, nor does it intelligently distribute heavy flows across CPU cores to balance the load. When the packet rates of individual heavy flows in the workload are within the processing capabilities of a single CPU core, RSS++ [35] can improve upon RSS by spreading the heavy flows across CPU cores. By measuring and adapting to the workload on each CPU core, RSS++ can scale throughput effectively with additional CPU cores, as has been shown in prior work [35, 63]. However, when the workload includes heavy flows whose (individual) packet rates exceed the processing capability of a single CPU core, and the number of such flows is smaller than the number of available CPU cores, simply spreading out the heavy flows across cores is insufficient to improve throughput with additional cores. This is the situation with the real packet workloads we tested (§4.1). Such situations also arise under attacks that force sharded packets to be handled on the same CPU core [43, 62]. It becomes necessary to process individual heavy flows using multiple cores, which is what SCR does.

Our results also show that RSS++ is not always better than RSS. Re-balancing load by migrating a flow shard across cores requires bouncing the cache line(s) containing the flow states across cores, an action that can degrade performance if done too frequently. On the other hand, rebalancing too infrequently may fail to mitigate skew across cores, since the per-shard load used to make CPU-balancing decisions may become stale relative to the current load.

Why does SCR scale better than other techniques? Figure 8 shows detailed performance metrics measured from Intel's

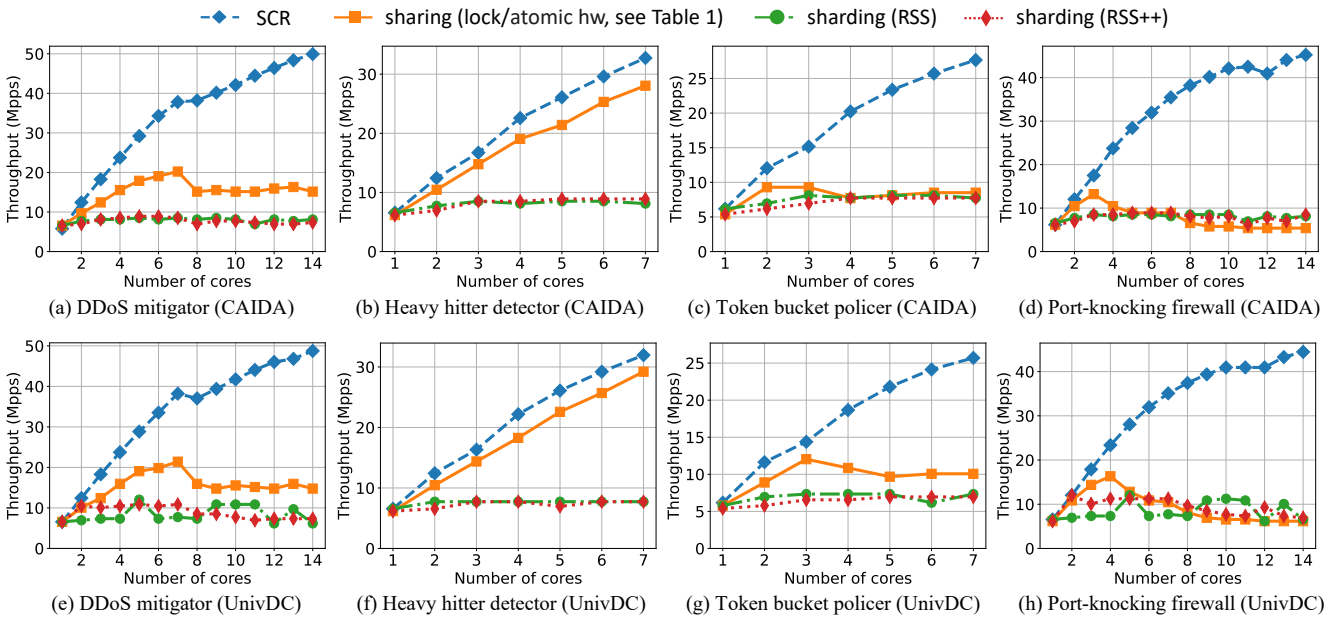


Figure 6: Throughput (millions of packets per second) of four stateful packet-processing programs implemented using state-compute replication (§3), shared state, and sharding (§2). Packet traffic is replayed from real data center and Internet backbone traces (§4.1).

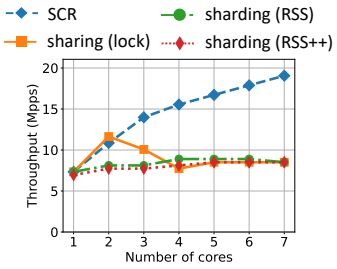


Figure 7: Throughput of TCP connection tracking parallelized using four techniques, SCR (§3), shared state, sharding with RSS, and sharding with RSS++ [35], on a hyperscalar data center trace (§4.1).

performance counter monitor (PCM [24]) and BPF profiling [14]. We measure the L2 cache hit ratios, instructions retired per cycle (IPC), and the program's computation latency (only the XDP portion, excluding the dispatch functionality in the driver), as the load offered to the system increases, when our token bucket policer is run across different numbers of cores (2, 4, or 7). The numbers show the averages for these metrics across the cores running the program. Error bars for IPC show the min and max values across cores. IPC is a meaningful metric to evaluate "CPU goodput" for eBPF/XDP programs: unlike high-speed packet processing frameworks like DPDK which poll the NIC and exhibit persistently high IPC [42], eBPF/XDP drivers adapt CPU usage to load through a mix of polling and interrupts.

Lock-based sharing in general suffers from lower L2 cache hit ratios ((a)–(c)) and higher latencies ((g)–(i)) due to lock and cache line contention across cores—a trend that holds as the offered load increases and also with additional cores

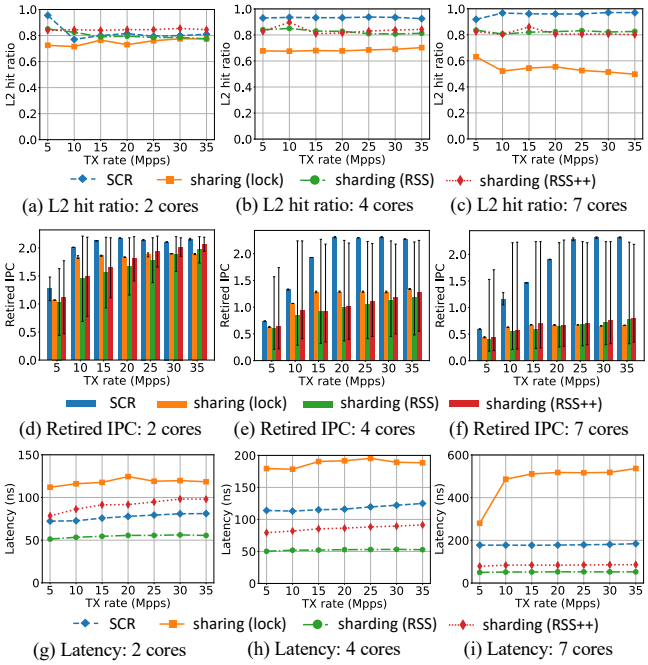
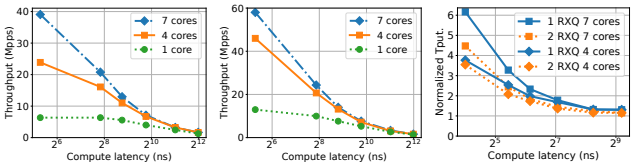


Figure 8: Hardware performance metrics drawn from Intel PCM while executing the token bucket program. As the offered load increases, we show the program's compute latency (measured purely for the XDP portion), the L2 hit ratio, and the number of instructions retired per CPU clock cycle (IPC), when the program is scaled to 2, 4, or 7 cores. Packet traffic is from a university data center (§4.1).



(a) Packets/sec (1 RXQ) (b) Packets/sec (2 RXQ) (c) Normalized to 1 core

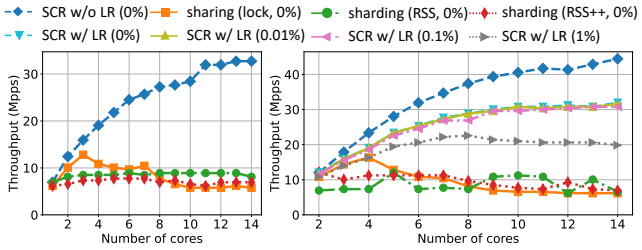
Figure 9: The throughput scaling of a stateless program using SCR, as the compute latency of the program varies but the dispatch latency remains constant ((a) and (b) in packets/second, and (c) normalized against single-core throughput at the same compute latency). As discussed in §3.1, the more the dispatch time dominates compute time, the more effective the multi-core scaling from SCR.

at the same offered load. As we might expect, IPC increases with the offered load ((d)–(f)), since the cores get busier with packet processing. While the sharding approaches (RSS and RSS++) effectively use the CPU with a high average IPC for 2 cores, their average IPC values drop significantly with additional cores, with large variation across cores (see error bars). This indicates a severe imbalance of useful CPU work across cores: Flow-affinity-based sharding approaches are unable to balance packet-processing load across cores, leaving some cores idle and others heavily used.

In contrast to sharing and sharding approaches, SCR has a consistently high IPC with more cores and higher offered loads. SCR has higher packet-processing latency ((g)–(i)) than RSS-sharding since it needs to process the history for each packet. RSS++ sometimes incurs higher compute latency than SCR due to its need to monitor per-shard load, which requires additional memory operations. SCR's more effective usage of the CPU cores results in better throughput scaling than RSS and RSS++ (Figure 6 (g)).

Limits to SCR scaling. SCR suffers from two kinds of scaling limitations. First, as discussed in §3.1, as the compute latency increases in comparison to the dispatch latency, the effectiveness of SCR's multi-core scaling reduces. This is because more time is spent "catching up" state, incurring significant duplicated work across CPU cores. We evaluate how the throughput of a stateless program varies as the compute latency of this program increases, shown both in packets/second (Figure 9a, Figure 9b) and normalized to the single-core throughput for that compute latency (Figure 9c). With a small compute latency (left of the graph), using N cores provides $\approx N\times$ throughput relative to a single core, but this relative benefit diminishes with increasing compute latency. Latency profiles of our benchmarks show compute latencies smaller than 70 nanoseconds (Table 4, Appendix A).

Second, SCR's attachment of histories to packets incurs non-negligible byte/second overheads. Adding to the number of bytes per packet increases L3 cache pressure due to higher DDIO cache occupancy [22] and incurs additional PCIe transactions and bandwidth [59]. Further, when packet histories are appended outside the NIC (*e.g.,* top-of-the-rack switch),



(a) Adding packet history externally (b) Impact of loss recovery

Figure 10: (a) The throughput of a token bucket policer on the university data center trace (§4.1), while truncating all packets in the trace to 64 bytes, with only SCR adding metadata to packets before feeding them to the NIC. (b) The throughput of a port-knocking firewall on the university data center trace. SCR is run with and without loss recovery (§3.4) at multiple packet loss rates.

Rows	LUT			Flip-flops	
	Usage	Logic	%	Usage	%
16	1045	646	0.060	2369	0.069
32	1852	1444	0.107	3158	0.091
64	2637	2229	0.153	4707	0.136
128	3390	2982	0.196	7786	0.226

Table 2: Sequencer resource usage after synthesis into the NetFPGA-PLUS reference switch and meeting timing at 340 MHz.

SCR may saturate the NIC earlier than other approaches. We compare SCR against the shared and sharded approaches, when SCR alone adds history metadata before packets are fed into the NIC while the packets for the latter approaches are truncated to 64 bytes. Figure 10 (a) shows the throughput of the token bucket program with the university data center trace. After 11 cores, the CPU is no longer the bottleneck for SCR. This prevents SCR from scaling to a higher packets/second throughput. Yet, SCR saturates at a throughput much higher than the other techniques.

Overheads of SCR's loss recovery handling. We evaluate how SCR's loss recovery algorithm impacts throughput with and without packet loss. Figure 10 (b) compares a version of SCR without incorporating the loss recovery algorithm, against a version that incorporates loss recovery at different artificially-injected random packet loss rates (0%, 0.01%, 0.1% and 1%). We also show the performance of existing scaling techniques (shared state, RSS, RSS++). The mere inclusion of the loss recovery algorithm impacts performance due to the additional logging operations. Moreover, SCR's throughput degrades with higher loss rate due to recovery-related synchronization (§3.4). However, SCR still outperforms and outscales existing multi-core scaling techniques.

4.3 Practicality of Sequencer Hardware

We integrated our Verilog module implementing the sequencer (§3.3.2) into the NetFPGA-PLUS [12] reference

Resource	Avg%	Resource	Avg%
Exact match crossbars	23.31%	SRAM	9.69%
VLIW instructions	9.11%	TCAM	0.00%
Stateful ALUs	93.75%	Map RAM	15.62%
Logical tables	23.96%	Gateway	23.44%

Table 3: Resource usage (average % across stages) of a Tofino implementation of the sequencer that uses as many stateful ALUs as possible to store packet history, amounting to 44 32-bit fields.

switch, which is clocked at a maximum frequency of 340 MHz with a 1024-bit wide data bus, providing a bandwidth of 348 Gbit/s. We use the Alveo U250 board, which contains 1728000 lookup tables (LUTs) and 3456000 flip-flops.

We synthesized our sequencer design with different numbers of memory rows (§3.3.2), corresponding to the size of the packet history (in number of packets). Each row is 112 bits long, enough to maintain a TCP 4-tuple and an additional 16-bit value (*e.g.,* a counter, timestamp, *etc.*) for each historic packet. Table 2 shows the resource usage. If each packet history metadata in the program is smaller than a row (112 bits), parallelizing across N cores requires N rows. For such programs, our design can meet timing (340 MHz) while scaling to 128 cores. The LUT and flip-flop hardware usage is negligible compared to the FPGA capacity at all row counts measured. We believe that our sequencer design may be simple and cheap enough to be added as an on-chip accelerator to a future NIC.

We have also implemented a stateful-register-based design of the sequencer on the Tofino programmable switch (§3.3.2). The resource usage of this design is shown in Table 3. Our implementation was designed to use as many stateful registers and ALUs as possible (our design uses 93% on average across stages) to hold the largest number of bits of packet history. Our design holds 44 32-bit fields, sufficient to parallelize (Table 1) the DDoS mitigator over 44 cores, the port-knocking firewall over 22 cores, the heavy hitter and token bucket over 9 cores, or the connection tracker over 5 cores. The small number of stateful ALUs on the platform, as well as the limit on the number of bits that can be read out from stateful memory into packet fields, restrict the Tofino sequencer from scaling to a larger number of CPU cores.

5 Related Work

Frameworks for network function performance. The problem of scaling out packet processing is prominent in network function virtualization (NFV), with frameworks such as split/merge [68], openNF [46], and Metron [54] enabling elastic scaling. There have also been efforts to parallelize network functions automatically [63] and designing data structures to minimize cross-core contention [47]. These efforts are flow-oriented, managing and distributing state at flow granularity. In contrast, SCR scales packet processing for a single flow.

General techniques for software parallelism. Among the canonical frameworks to implement software parallelism [57], our scaling principles are most reminiscent of Single Program Multiple Data (SPMD) parallelism, with the program being identical on each core but the data being distinct. The sequencer in SCR makes the data distinct for each core.

Parallelizing finite state machines. A natural model of stateful packet processing programs is as finite state automata (the state space is the set of flow states) making transitions on events (packets). There have been significant efforts taken to parallelize FSM execution using speculation [65, 66] and data parallelism [58]. In contrast, SCR exploits replication.

Parallel network software stacks. There has been recent interest in abstractions and implementations that take advantage of parallelism in network stacks, for TCP [55, 72] and for end-to-end data transfers to/from user space [39]. SCR takes a complementary approach, using replication rather than decomposing the program into smaller parallelizable computations.

6 Conclusion

It is now more crucial than ever to investigate techniques to scale packet processing using multiple cores. This paper presented state-compute replication (SCR), a principle that enables scaling the throughput of stateful packet-processing programs monotonically across cores by leveraging a packet history sequencer, even under realistic skewed packet traffic.

Acknowledgments

The work was supported in part by the U.S. National Science Foundation (NSF) grants CNS #1910796, CAREER #2340748, CNS #2008048, FMitF #2019302, and FMitF #2422076, the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, the partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"), and gifts from the Network Programming Initiative (NPI) and Xilinx. We thank our shepherd Luis Pedrosa for his thoughtful comments on the final versions of this paper and the anonymous NSDI reviewers for their revision feedback. We are grateful to David Walker, Jennifer Rexford, Sanjay Rao, and Santosh Nagarakatte for helpful feedback on previous versions of this paper. We also appreciate the support of the Rutgers Laboratory for Computer Science Research (LCSR) staff, Kevin Conover, Tim Hayes, and Chuck Hedrick, for their thoughtful support of research infrastructure.

References

[1] Intel IPU. [Online, Retrieved Feb 21, 2023.] https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html.

[2] Linux Socket Filtering aka Berkeley Packet Filter (BPF). [Online, Retrieved Nov 05, 2020.] https://www.kernel.org/doc/Documentation/networking/filter.txt.

[3] NVIDIA BlueField DPU. [Online, Retrieved Feb 21, 2023.] https://www.nvidia.com/en-us/networking/products/data-processing-unit.

[4] Receive Side Scaling. [Online, Retrieved Feb 21, 2023.] https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[5] Benchmarking Methodology for Network Interconnect Devices. [Online, Retrieved Sep 17, 2023.] https://www.rfc-editor.org/rfc/rfc2544, 1999.

[6] IEEE 802.1Qbb – Priority-based Flow Control. [Online, Retrieved May 02, 2024.] https://1.ieee802.org/dcb/802-1qbb/, 2011.

[7] How to set up Intel Ethernet Flow Director. [Online, Retrieved Sep 17, 2023.] https://www.intel.com/content/www/us/en/developer/articles/training/setting-up-intel-ethernet-flow-director.html, 2017.

[8] Open-sourcing Katran, a scalable network load balancer. [Online, Retrieved Sep 17, 2023.] https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/, 2018.

[9] A kernel patch to support RSS++. [Online, Retrieved Apr 29, 2024.] https://github.com/rsspp/linux/commit/4e09cf8be6ac5b0a06cc5b92c62f758f29e3b6aa, 2019.

[10] Concurrency management in eBPF. [Online, Retrieved Sep 17, 2023.] https://lwn.net/Articles/779120/, 2019.

[11] The CAIDA UCSD Anonymized Internet Traces - 2019. [Online, Retrieved Sep 17, 2023.] https://www.caida.org/catalog/datasets/passive_dataset, 2019.

[12] NetFPGA-PLUS. [Online, Retrieved Sep 17, 2023.] https://github.com/NetFPGA/NetFPGA-PLUS, 2021.

[13] Nvidia ConnectX-6 DX. [Online, Retrieved Apr 14, 2024.] https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/ConnectX-6-Dx-Datasheet.pdf, 2021.

[14] Measuring BPF performance: Tips, tricks, and best practices. [Online, Retrieved Sep 17, 2023.] https://developers.redhat.com/articles/2022/06/22/measuring-bpf-performance-tips-tricks-and-best-practices, 2022.

[15] BPF helpers manual page. [Online, Retrieved Sep 17, 2023.] https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html, 2023.

[16] BPF maps. [Online, Retrieved Sep 17, 2023.] https://www.kernel.org/doc/html/latest/bpf/maps.html, 2023.

[17] Broadcom Trident 4: BCM56880 Series. [Online, Retrieved Jul 22, 2023.] https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, 2023.

[18] Built-in Functions for Memory Model Aware Atomic Operations. [Online, Retrieved Sep 17, 2023.] https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/_005f_005fatomic-Builtins.html, 2023.

[19] Checksum Offloads. [Online, Retrieved Sep 17, 2023.] https://www.kernel.org/doc/html/latest/networking/checksum-offloads.html, 2023.

[20] ConnectX-7 400G adapters. [Online, Retrieved May 1, 2024.] https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-7-datasheet, 2023.

[21] Data Plane Development Kit. [Online, Retrieved Jul 22, 2023.] https://www.intel.com/content/www/us/en/developer/topic-technology/networking/dpdk.html, 2023.

[22] Intel Data Direct I/O technology. [Online, Retrieved Sep 17, 2023.] https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, 2023.

[23] Intel Tofino. [Online, Retrieved Jul 22, 2023.] https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, 2023.

[24] Introduction to Intel Performance Counter Monitor (PCM). [Online, Retrieved Sep 17, 2023.] https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html, 2023.

[25] Legacy __sync Built-in Functions for Atomic Memory Access. [Online, Retrieved Sep 17, 2023.] https://gcc.gnu.org/onlinedocs/gcc-13.2.0/gcc/_005f_005fsync-Builtins.html, 2023.

[26] Open Tofino: P4 Intel Tofino Native Architecture - Public version. [Online, Retrieved Sep 17, 2023.] https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf, 2023.

[27] Pensando infrastructure accelerators. [Online, Retrieved Jul 22, 2023.] https://www.amd.com/en/accelerators/pensando, 2023.

[28] Port knocking. [Online, Retrieved Jul 22, 2023.] https://help.ubuntu.com/community/PortKnocking, 2023.

[29] Segmentation Offloads. [Online, Retrieved Sep 17, 2023.] https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html, 2023.

[30] TCP Offload Engine: Chelsio Communications. [Online, Retrieved Sep 17, 2023.] https://www.chelsio.com/nic/tcp-offload-engine/, 2023.

[31] SCR experimental scripts and software. [Online, Retrieved Oct 03, 2024.] https://github.com/smartnic/bpf-profile, 2024.

[32] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. Understanding host interconnect congestion. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2022.

[33] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[34] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-Speed NICs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[35] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. Rss++: Load and state-aware receive side scaling. In *Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, 2019.

[36] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *Internet Measurement Conference (IMC)*, 2010.

[37] Daniel Borkmann and Martynas Pumputis. K8s Service Load Balancing with BPF & XDP. [Online. Retrieved Jan 23, 2021.] https://linuxplumbersconf.org/event/7/contributions/674/attachments/568/1002/plumbers_2020_cilium_load_balancer.pdf, 2020.

[38] Pat Bosshart et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM*, 2013.

[39] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: Disaggregating the host network stack. In *ACM SIGCOMM*, 2022.

[40] Arthur Chiao. Connection Tracking (conntrack): Design and Implementation Inside Linux Kernel. [Online, Retrieved Sep 17, 2023.] https://arthurchiao.art/blog/conntrack-design-and-implementation/, 2020.

[41] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NIC-Driven Tail-Aware Balancing of us-Scale RPCs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[42] 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 Symposium on Operating Systems Principles (SOSP)*, 2009.

[43] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[44] Arthur Fabre. L4drop: Xdp ddos mitigations. [Online, Retrieved Jul 25, 2023.] https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/.

[45] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[46] 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*, 2014.

[47] Massimo Girondi, Marco Chiesa, and Tom Barbette. High-speed connection tracking in modern servers. In *IEEE International Conference on High Performance Switching and Routing (HPSR)*, 2021.

[48] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *ACM SIGCOMM*, 2016.

[49] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 2015.

[50] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[51] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[52] Jonathan Corbet. Accelerating networking with AF_XDP. [Online. Retrieved Jan 20, 2021.] https://lwn.net/Articles/750845/, 2018.

[53] Pravein Govindan Kannan, Nishant Budhdev, Raj Joshi, and Mun Choon Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[54] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV service chains at the true speed of the underlying hardware. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[55] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *ACM EuroSys*, 2019.

[56] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[57] Samuel Midkiff. *Automatic parallelization: an overview of fundamental compiler techniques*. Springer Synthesis Lectures on Computer Architecture, 2012.

[58] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-Parallel Finite-State Machines. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[59] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *ACM SIGCOMM*, 2018.

[60] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM*, 2021.

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

[62] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. Automated synthesis of adversarial workloads for network functions. In *ACM SIGCOMM*, 2018.

[63] Francisco Pereira, Fernando M.V. Ramos, and Luis Pedrosa. Automatic parallelization of software network functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[64] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[65] Junqiao Qiu, Xiaofan Sun, Amir Hossein Nodehi Sabet, and Zhijia Zhao. Scalable FSM Parallelization via Path Fusion and Higher-Order Speculation. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[66] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. Enabling Scalability-Sensitive Speculative Parallelization for FSM Computations. In *International Conference on Supercomputing*, 2017.

[67] Quentin Monnet. Optimizing BPF: Smaller Programs for Greater Performance. [Online. Retrieved Jan 20, 2021.] https://www.netronome.com/blog/optimizing-bpf-smaller-programs-greater-performance/, 2020.

[68] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System support for elastic execution in virtual middleboxes. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[69] Luigi Rizzo. Netmap: a novel framework for fast packet I/O. In *USENIX annual technical conference (ATC)*, 2012.

[70] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[71] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. A case for spraying packets in software middleboxes. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2018.

[72] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[73] Nikita V. Shirokov. XDP: 1.5 years in production. Evolution and lessons learned. In *Linux Plumbers Conference*, 2018.

[74] Shinae Woo and KyoungSoo Park. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling. Technical report, KAIST, 2020.

[75] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of internet flow rates. In *ACM SIGCOMM*, 2002.

Application	t	c_2	d	c_1
DDoS mitigator	126	13	101	25
Heavy hitter monitor	138	17	105	32
Token bucket policer	153	22	102	51
Port-knocking firewall	128	15	101	27
TCP connection tracking	140	39	71	69

Table 4: The throughput model parameters (in nanoseconds) for packet-processing applications we evaluated.

Appendixes

The appendixes include supporting material that has not been peer-reviewed.

A Predicting Throughput

This section provides a model that predicts the throughput of SCR (§3.2) for any number of cores, given latency measurements made with one and two cores at low load. We also check the agreement between the empirically-measured throughput and our model.

Suppose a system has k cores, where each core can dispatch a single packet in d cycles, and run a packet-processing program that computes over a single packet in $c = c_1 + (k-1) \times c_2$ cycles, where c_1 is the time required to process the current packet and c_2 the time for a state update using one item of packet history. The time c_2 is smaller than c_1, as the state transition is a fragment of the computation extracted from the program which processes one packet in entirety. For each piggybacked packet, the total processing time is $d + c_1 + (k-1) \times c_2$. We define $t \triangleq d + c_1$, the time to process one packet including the dispatch and the program computation. When t dominates the state-computation time (*i.e.,* $t \gg c_2$), with k cores, the total rate at which externally-arriving packets can be processed is $k \times \frac{1}{t+(k-1)\times c_2} \approx k/t$. Table 4 lists the parameters we measured for packet-processing applications we evaluated. It shows that $t \approx 3.6 - 9.9 \times c_2$. Hence, it is possible to scale the packet-processing rate linearly with the number of cores k.

We applied the parameters in Table 4 to the throughput model and compared the predicted throughput to the actual throughput. Figure 11 shows that they match well.

B Loss Recovery Algorithm

This section provides the detailed pseudocode and a proof of correctness of the packet loss recovery algorithm outlined in §3.4.

Before we start the correctness proof of loss recovery algorithm, we define the following notations.

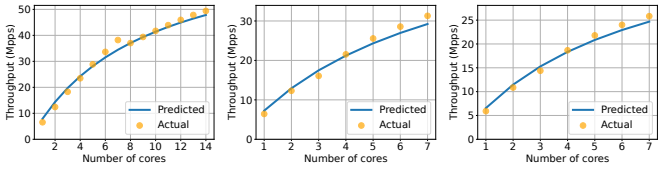
1. sp: an SCR packet. sp_i: the i^{th} SCR packet sent from the sequencer to a core (please refer to §3.3.1 for more details).

Algorithm 1 SCR loss recovery

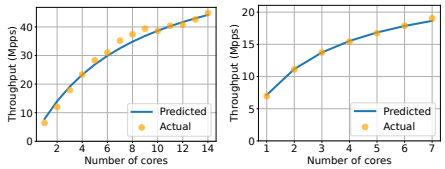
Input: the received SCR pkt (sp), current core (c)

```
1:  Initialize:
        max[c] ← 0, ∀i : log[c][i] ← NOT_INIT
2:  function scr_loss_recovery(sp, c)
3:      maxseq ← sp.seq_num
4:      minseq ← max(1, maxseq − N + 1)
5:      for i ← max[c] + 1 to maxseq do
6:          if i < minseq then
7:              log[c][i] ← LOST
8:              handle_loss_recovery(i, c)
9:          else
10:             history ← get history of i from sp
11:             log[c][i] ← history
12:         end if
13:     end for
14:     max[c] ← maxseq
15: end function

16: function handle_loss_recovery(i, c)
17:     C_others ← C \ {c}
18:     C_lost ← ∅
19:     while true do
20:         for each c′ ∈ C_others do
21:             if log[c′][i] is NOT_INIT then
22:                 continue
23:             else if log[c′][i] is LOST then
24:                 C_lost ← C_lost ∪ {c′}
25:                 if C_lost = C_others then return
26:                 end if
27:             else
28:                 history ← log[c′][i]
29:                 update state using history
30:                 return
31:             end if
32:         end for
33:     end while
34: end function
```



(a) DDoS mitigation (b) Heavy hitter detector (c) Token bucket policer



(d) Port-knocking firewall (e) TCP connection tracking

Figure 11: Predicted and actual throughput (§4.1) in millions of packets per second (Mpps) of five stateful packet-processing programs implemented using SCR (§3). The workloads of (a)-(d) and (e) are from a university data center and a hyperscalar data center (§4.1) separately.

2. p: a regular packet. p_i: the i^{th} regular packet received by the sequencer (in original or as part of the packet history, please refer to §3.3.1 for more details).
3. C: the collection of all cores.

We want to prove that no core will be deadlocked by loss recovery, *i.e.,* every core will *start processing* (execute line 6) and *finish processing* (finish executing line 6-12) every regular packet, under the assumptions that (1) each core will receive at least one SCR packet after packet loss, (2) we have infinite memory, and (3) packet number monotonically increases.

Theorem 1. *For any regular packet p_i, if every core has received p_j ($j \geq i$), every core will finish processing p_i.*

Proof. We will first prove that any core will process p_1 to p_i in order. The order of packets to process follows the order at line 5 (from $max[c]+1$ to $maxseq$), *i.e.,*, after a core finishes processing p_k, it will start processing p_{k+1}. Each core will be triggered to process all packets including p_1 to p_i, since each core has received p_j ($j \geq i$),

Given the order of packets to process, we now prove all cores can finish processing p_i. If $i = 1$, all cores will start processing p_1 (line 6) and then finish processing p_1 (Lemma 1). If $i > 1$, according to the order of a single core processing packets and Lemma 1, we get the induction hypothesis that if all cores have started processing p_k, all cores will finish processing p_k and start processing p_{k+1}. Using induction hypothesis for i times, all cores will start and then finish processing p_i. □

Lemma 1. *For any regular packet p_i, if all cores have started processing p_i, then all cores will finish processing p_i.*

Proof. For any core c, no matter p_i is lost or received, c will finish processing it.

If p_i is received by c, after c updates $history[i]$ in its log (line 10-11), c finishes processing p_i.

If p_i is lost at c (detected at line 6), c will wait for other cores to update p_i in their logs until c gets $history[i]$ or confirm p_i is lost at all the other cores (line 19-33). c will not be deadlocked in waiting, since p_i will be updated to $history[i]$ or *LOST* in the logs of all cores which have started processing p_i (if line 6 is executed for p_i, line 7 or line 11 will be executed). □

In a practical implementation, logs are finite and sequence numbers wrap around. We handle these concerns with a sufficiently large log and a sequence space. Our current implementation uses the values 1,024 and 842,185 for the aforementioned two quantities (respectively).

C SCR-Aware Multi-Core Programming

Consider a packet-processing program developed assuming single-threaded execution on a single CPU core. The question we tackle in this subsection is: how should the program be changed to take advantage of multi-core scaling with state-compute replication? We walk through the process of adapting a program written in the eBPF/XDP framework [50], but we believe it is conceptually similar to adapt programs written in other frameworks such as DPDK.

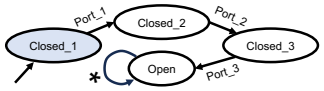


Figure 12: A state machine for a simple port-knocking firewall.

We describe the program transformations necessary for SCR through a running example. Suppose we have a port-knocking firewall [28] with the state machine shown in Figure 12. The program runs a copy of this state machine per source IP address. If a source transmits IPv4/TCP packets with the correct sequence of TCP destination ports, then all further communication is permitted from that source. All other packets are dropped. Any transition not shown in the figure leads to the default `CLOSED_1` state, and only the `OPEN` state permits packets to traverse the firewall successfully. A simplified XDP implementation of this single-threaded firewall is shown below.

```
/* Definition of program state */
struct map states {
  /* assume we define a dictionary with keys
     as source IP addresses and values as firewall
     states among CLOSED_{1, 2, 3} and OPEN. */
}

/* State transition function. See Figure 12  */
int get_new_state(int curr_state, int dport) {
  /* A function that implements the state machine for
     the port knocking firewall. */
  if (curr_state == CLOSED_1 && dport == PORT_1)
    return CLOSED_2;
```

```
  if (curr_state == CLOSED_2 && dport == PORT_2)
    return CLOSED_3;
  if (curr_state == CLOSED_3 && dport == PORT_3)
    return OPEN;
  if (curr_state == OPEN)
    return OPEN;
  return CLOSED_1;
}

/* The main function */
int simple_port_knocking(...) {
  /* Assume the packet is laid out as a byte array
     starting at the address pkt_start. Suppose the
     packet is long enough to include headers up to
     layer 4. First, parse IPv4/TCP pkts. */
  struct ethhdr* eth = pkt_start; // parse Ethernet
  int l3proto = eth->proto; // layer-3 protocol
  int off = sizeof(struct ethhdr);
  struct iphdr* iph = pkt_start + off;
  int l4proto = iph->protocol; // layer-4 protocol
  if (l3proto != IPv4 || l4proto != TCP)
      return XDP_DROP; // drop non IPv4/TCP pkts
  int srcip = iph->src; // source IP addr
  off += sizeof(struct iphdr);
  struct tcphdr* tcp = pkt_start + off;
  int dport = tcp->dport; // TCP dst port

  /* Extract & update firewall state for this src. */
  int state = map_lookup(states, srcip);
  int new_state = get_new_state(state, dport);
  map_update(states, srcip, new_state);

  /* Final packet verdict */
  if (new_state == OPEN)
      return XDP_TX; // allow traversal
  return XDP_DROP; // drop everything else
}
```

The program's state is a key-value dictionary mapping source IP addresses to an automaton state described in Figure 12. The function `get_new_state` implements the state transitions. The main function, `simple_port_knocking` first parses the input packet, dropping all packets other than IPv4/TCP packets. Then the program fetches the recorded state corresponding to the source IP on the packet, and performs the state transition corresponding to the TCP destination port. If the final state is `OPEN`, all subsequent packets of that source IP may traverse the firewall to the other side. All other packets are dropped.

To enable this program to use state-compute replication across cores, this program should be transformed in the following ways. We believe that these transformations may be automated by developing suitable compiler passes, but we have not yet developed such a compiler.

(1) *Define per-core state data structures and per-packet metadata structures.* First, the program's state must be replicated across cores. To achieve this, we must define per-core state data structures that are identical to the global state data structures, except that they are not shared among CPU cores. Packet-processing frameworks provide APIs to define such per-core data structures [16].

Additionally, we must define a per-packet metadata structure that includes any part of the packet that is used by the program—through either control or data flow—to update the state corresponding to that packet. For the port-knocking firewall, the per-packet metadata should include the `l3proto`, `l4proto`, `srcip`, and `dport`.

The data structures that maintain packet history on the sequencer correspond to this per-packet metadata (§3.3).

(2) Fast-forward the state machine using the packet history. The SCR-aware program must prepend a loop to "catch up" the state machine for each packet missed by the CPU core where the current packet is being processed. By leveraging the recent history piggybacked on each packet, at the end of this loop, the CPU core has the most up-to-the-packet state.

```
/* Assume the pointer 'data' locates where the
   per-pkt metadata begins in the byte array of
   the packet (Figure 4a ). Suppose 'index'
   is the offset of the earliest packet §3.3.2 ,
   and NUM_META is the number of packets in the
   piggybacked history.
*/
int l3proto, l4proto, srcip, dport, i, j;
for (j = 0; j < NUM_META; j++) {
  i = (index + j) % NUM_META; // ring buffer
  struct meta *pkt = data + i * sizeof(meta);
  l3proto = pkt->l3proto;
  l4proto = pkt->l4proto;
  srcip   = pkt->srcip;
  dport   = pkt->dport;
  if (l3proto != IPv4 || l4proto != TCP)
     continue; // no state txns or pkt verdicts
  /* Update state for this srcip and dport: */
  /* map_lookup; get_new_state; map_update. */
  /* Note: No pkt verdicts for historic pkts. */
}
pkt_start = data + NUM_META * sizeof(struct meta)
              + sizeof(index);
```

A few salient points about the code fragment above. First, the semantics of the ring buffer of packet history (§3.3) are implemented by looping over the packet history metadata starting at offset `index` rather than at offset 0. The decision to implement the ring buffer semantics in software makes the hardware significantly easier to design, since only a small part of the hardware data structure needs to be updated for each packet (§3.3.2). Second, the loop must implement appropriate control flow before the state update to ensure that only packets that should indeed update the flow state do. Note that the metadata includes parts of the packet that are not only the data dependencies for the state transition (`srcip, dport`) but also the control dependencies (`l3proto, l4proto`). Third, no packet verdicts are given out for packets in the history: we want the program to return a judgment for the "current"

packet, not the historic packets used merely to fast-forward the state machines. Finally, the code fragment conveniently adjusts `pkt_start` to the position in the packet buffer (Figure 4a) corresponding to where the "original" packet begins. The rest of the original program—unmodified—may process this packet to completion and assign a verdict.

What is excluded in our code transformations is also crucial. This program avoids locking and explicit synchronization, despite the fact that it runs on many cores, even if there is global state maintained across all packets.

With these transformations, in principle, a packet-processing program is able to scale its performance using state-compute replication across multiple cores.