Using High-throughput Pipelines to Parallelize Stateful Packet Processing

Qiongwen Xu*
Rutgers University

Songyuan Zhang*
Rutgers University

Sebastiano Miano
Queen Mary University of London

Anirudh Sivaraman
New York University

Gianni Antichi
Politecnico di Milano

Srinivas Narayana
Rutgers University

Background. Software packet processing must be high throughput, low latency, and CPU-efficient. Over time, the academic and industry networking communities have been improving the performance of packet processing on endpoints by removing software from the critical packet-processing path. In this work, we focus on *programmable NIC offloads*, the ability to process packets on programmable components of emerging network interface cards (NICs) such as Mellanox BlueField, Fungible DPUs, Intel IPUs, and Pensando [1, 2]. These emerging NIC platforms contain both general-purpose programmable CPU cores (*e.g.,* ARM) as well as custom accelerators, notably high-speed packet-processing pipelines.

Offloading middlebox packet-processing tasks, such as implementing a Maglev load balancer or a port-knocking firewall, to programmable NICs is not easy. Packet processing pipelines [4] achieve high throughput by processing several packets simultaneously across stages while running at a high clock rate. However, it is challenging to support complex stateful operations, *i.e.*, maintaining memory across packets [6], say for monitoring or control. The first reason is that any pipeline stage that implements a stateful operation has to read, modify, and write its output back in a very short time (*e.g.,* one clock cycle) to ensure high pipeline throughput. This limits the expressiveness of the operation. Second, high clock rates limit the density (and hence sizes) of the memories used to maintain state across packets. In contrast, the general-purpose cores on NICs support expressive instruction sets, and are augmented with large memories. However, each core is designed to be smaller and slower than server CPU cores, due to limitations in the NIC power budget and form factor.

This project seeks to combine the speed of pipelines with the expressiveness and statefulness of general-purpose cores.

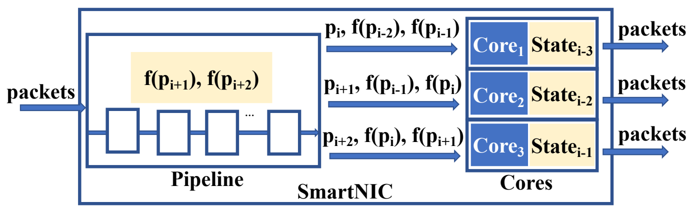
Motivation. Since a pipeline can process packets at line rate, the packet-processing performance of a programmable NIC is limited by its cores. However, a single general-purpose on-NIC core cannot meet line rate while running complex programs. It is natural to ask how to leverage *multiple cores* to process packets faster. To use multiple cores, prior work uses *flow affinity*, *i.e.*, assigning packets accessing the same state in memory (we call such groups of packets *flows*) to the same core, which avoids different cores contending to access the same memory. For example, receive-side scaling (RSS) is typically configured to hash packet fields, and choose a fixed CPU core to process all packets with a fixed hash. However, under hash collisions of heavy flows (known to occur frequently in heavy-tailed networking workloads, *e.g.,* [3]), processing cores are easily saturated. Hence, the achieved tail latency is bottlenecked by that of a single (slow) core.

Our Solution. In networking, packet spraying is a common technique to evenly spread across load network paths. We seek to adopt packet spraying for stateful packet processing code. However, to enable cores to process overlapping sets of flows, we should completely remove shared state across cores. We show that with a little help from the pipeline available on the programmable NIC, packet spraying may enable running stateful applications *with zero cross-core synchronization* for any distribution of packets to flows.

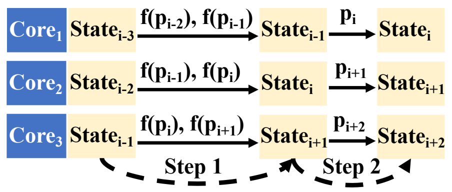
Our idea is to program the pipeline to spray packets in a round-robin fashion across cores. Each core runs an *independent, shared-nothing* copy of the application, with the application state (*e.g.,* TCP connection state) fully replicated in its local memory. In addition to the sprayed packet, the pipeline also piggybacks sufficient information on the sprayed packet to help the recipient core reconstruct the application's state with respect to the packets that the core missed. Instead of synchronizing state explicitly across cores, synchronization is performed implicitly by using the pipeline as a *reliable sequencer of all packets*, updating the replicated state machine running on each core.

An example. Consider Fig. 1. A pipeline and three cores are used to process packets here. As shown in Fig. 1a, the pipeline sprays packets (*i.e.*, p_i, p_{i+1}, p_{i+2}) in a round-robin fashion across cores (*i.e.*, $core_1, core_2, core_3$). Further, the pipeline stores a small amount of state shared across all packets. Specifically, if there are n cores, the pipeline stores the recent packet history consisting of the packet fields from the last n packets which are relevant to evolving the application-level state. Prior work on circular buffers implemented on switches [5] has shown that it is feasible to update packet histories at line rate on pipelines. Note that this packet history is updated only by the pipeline and is never written to by the cores. In

*Students. Contact: qx51@cs.rutgers.edu, songyzhang@hotmail.com, s.miano@qmul.ac.uk, anirudh@cs.nyu.edu, gianni.antichi@polimi.it, srinivas.narayana@rutgers.edu



(a) The pipeline stores relevant packet fields and sends packets along with fields on the packets each core missed to cores in a round-robin fashion.


$Core_1$ $State_{i-3}$ $f(p_{i-2}), f(p_{i-1})$ $State_{i-1}$ p_i $State_i$
$Core_2$ $State_{i-2}$ $f(p_{i-1}), f(p_i)$ $State_i$ p_{i+1} $State_{i+1}$
$Core_3$ $State_{i-1}$ $f(p_i), f(p_{i+1})$ $State_{i+1}$ p_{i+2} $State_{i+2}$
Step 1
Step 2

(b) Each core leverages the missing packet fields to process the sprayed packet independently.

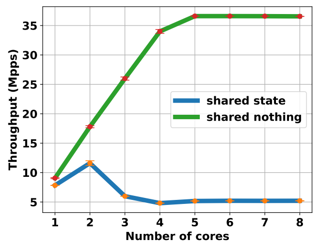
Figure 1: Overview of the shared-nothing approach. p_i is the i^{th} packet received by the smartNIC, $f(p_i)$ is state-computation-relevant fields from p_i, and $State_i$ is the program state value updated using packets $p_1, ..., p_i$ in order.

the example in Fig. 1(a), the packet history supplied to $core_1$ processing packet p_i is $f(p_{i-2}), f(p_{i-1})$. Each core updates its local application state by implementing *vector packet processing* on n packets rather than just the one packet sprayed to it. As shown in Fig. 1(b), before $core_1$ processes p_i, it will first process $f(p_{i-1}), f(p_{i-2})$. Since each core misses at most $n-1$ packets in round-robin spraying, it can "fast-forward" its application state to the most updated values using a packet history with a known, bounded size of at most $n-1$. This solution will always produce *correct* state and packet-level outcomes on each core.

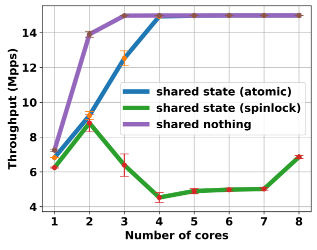
There are two key performance concerns. First, piggybacking packet histories of size n (determined by the available parallelism across cores or threads) may significantly increase the bandwidth requirements between the pipeline and each core. Fortunately, bandwidth within a programmable NIC is plentiful. Second, on the surface, overlapping vector computation across n packets appears wasteful: each core processes not just the packet it was sprayed, but also packets that all the cores were sprayed in the last round. However, packet-processing is often CPU-bottlenecked by *per-packet* (not per-byte) work, where 'packet' corresponds to a (physical) packet moved into and out of the system. Adding redundant computation corresponding to the (logical) packet history while moving one (physical) packet through the system does not significantly slow down the system.

Implementation and Preliminary Evaluation Result. We implemented two stateful applications developed in the eBPF framework, a stateful port-knocking firewall and a heavy hitter detector, separately using the shared-nothing and the shared-state (with fine-grained locking) approach. More specifically, for the heavy hitter application, we implemented two shared-state versions, one using a spin lock and another using hardware-optimized transactional memory operations.

To evaluate how throughput scales with cores, we use two servers on CloudLab to set up a high-speed packet generator using TRex and a Device Under Test (DUT). Each server has 10-core Intel Broadwell (E5-2640v4) 2.4 GHz processors with a PCIe 3.0 bus and 64 GB of memory, and is equipped with two Mellanox ConnectX-4 25Gbps NICs. We attach each packet-processing benchmark to the network device driver on the DUT and use the packet generator to send packets (the missing packet fields is attached in the packet) to the DUT at line rate. The packets will be processed by the benchmark and forwarded back to the packet generator. We measure the RX rate (*i.e.*, throughput) on the packet generator. Fig. 2 shows the throughput results. For both applications, our approach can reach the line rate with fewer cores used to process packets. Moreover, the throughput of our approach linearly increases with cores for both applications.



(a) Firewall using port-knocking



(b) Heavy hitter detection

Figure 2: Throughput (million packets per second) of two stateful applications using the shared-state and the shared-nothing approach. Error bars denote standard deviations.

Ongoing work. We are (1) developing a compiler to automate the transformation of code written as a single thread to sharing nothing across cores; and (2) automating the generation of P4 pipeline code to piggyback application-specific packet histories on each sprayed packet.

References

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

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

[3] Mohammad Al-Fares et al. Hedera: dynamic flow scheduling for data center networks. In *NSDI*, 2010.

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

[5] Pravein Govindan Kannan et al. Debugging transient faults in data centers using synchronized network-wide packet histories. In *NSDI*, 2021.

[6] Anirudh Sivaraman et al. Packet transactions: High-level programming for line-rate switches. In *SIGCOMM*, 2016.