# AppSteer: Framework for Improving Multicore Scalability of Network Functions via Application-aware Packet Steering


Ashwin Kumar
*IIT Bombay*
India
ashkumar@cse.iitb.ac.in

Rajneesh Katkam
*IIT Bombay*
India
rkatkam@cse.iitb.ac.in

Pranav Chaudhary
*IIT Bombay*
India
pchau@cse.iitb.ac.in

Priyanka Naik
*IBM Research*
India
priyanka.naik@ibm.com

Mythili Vutukuru
*IIT Bombay*
India
mythili@cse.iitb.ac.in



***Abstract*—Efforts to improve multicore scalability of network functions (NFs) have traditionally focused on making network stacks scalable via partitioning TCP/IP data structures into per-core slices and ensuring flow-to-core affinity, leading to elimination of locking in the network stack while processing an incoming packet. But the above techniques fail to eliminate locking in NFs which store state at the granularity of an application-layer key that does not map to a TCP/IP flow, e.g., NFs in the 5G packet core that store state at the granularity of a mobile subscriber/user, where requests from a user could arrive over multiple flows, or requests from multiple users can arrive on a single flow. Prior work does not allow steering all traffic of a particular user to the same core for such NFs. This paper presents AppSteer, a framework that enables application-aware steering of incoming requests to cores for NFs running on the Linux kernel, in order to localize the requests of a given application-layer entity (e.g., mobile user) to a single core. NFs running over AppSteer can then partition their state into per-core slices and access it in a lockfree manner, leading to better multicore scalability. We evaluate AppSteer by building lockfree versions of production-grade 5G core NFs running on top of AppSteer and show that they have 15–18% higher throughput at 16 cores when compared to their locking-based counterparts.**



***Index Terms*—packet steering, multicore scalability**


## I. Introduction

With advances in chip making technology allowing for a large number of processors to be placed on a single chip, there is an increasing focus on designing multicore scalable software (i.e., software whose performance scales with increasing cores), to take advantage of the parallelism available in modern CPUs [1]–[6]. This paper considers the multicore scalability of networked applications, also known as network functions (NFs), which are applications that receive network traffic over well-defined external interfaces and process this traffic as per the specified functional behavior of the NF. Most software NFs are built as multi-threaded packet processing applications that run over the multiple cores available in modern commodity servers, either baremetal or inside VMs/containers in a cloud. To improve performance, NFs are scaled horizontally by adding more replicas, and vertically by adding more cores to each replica. One important factor that limits the multicore scalability of such NFs is contention for locks that protect access to shared data structures, both within the userspace processing of the NF as well as inside the network stack.

Prior work on improving multicore scalability of NFs [7]–[14] has mainly considered the problem of reducing lock contention inside the network stack. Modern computer systems distribute the interrupt load of incoming traffic across multiple cores based on the hash of the connection 4-tuple, e.g., via mechanisms like Receive Side Scaling (RSS [15]). Modern multicore-scalable network stacks (often running on kernel bypass packet I/O mechanisms like DPDK [16]) then ensure that the network stack and userspace application processing of a transport layer connection is localized to the same core on which the traffic initially lands, by partitioning various TCP/IP data structures, e.g., accept queues, into per-core slices. This localization reduces the contention for locks on shared network stack data structures across cores, improves CPU cache locality, leading to better multicore scalability.

However, the above techniques do not eliminate locking needed to safely access state shared across the multiple application threads in userspace. The above techniques may reduce lock contention at the NF userspace only if the NF maintains state at the granularity of TCP/IP flows (eg. NATs, load balancers). However, many stateful NFs may store state at the granularity of an application level entity that does not map to TCP/IP flows, e.g., at the granularity of application layer keys in a key-value store. In such NFs, requests for a given application layer entity/key could land on the NF over multiple transport layer connections, or requests for multiple entities/keys could arrive over a single (persistent, long running) transport layer connection. Therefore, localizing the transport layer connection to a single core will not eliminate locking needed to access state at the NF.

An important set of NFs that do not maintain state at the granularity of TCP/IP flows are the NFs in the mobile packet core of a telecom network. The packet core consists of control plane components that implement various signaling procedures (e.g., registering a user to the network, setting up data transfer sessions with suitable QoS) and data plane components that forward user traffic between the mobile subscriber (user) and other external networks. While the various components in a mobile network were traditionally built as custom packet-processing hardware, recent generations are seeing a greater push towards softwarization. The 5G packet core [17] is composed of several software NFs running on commodity hardware, in accordance with the principle of Network Function Virtualization (NFV).

The NFs in the mobile packet core maintain state at the

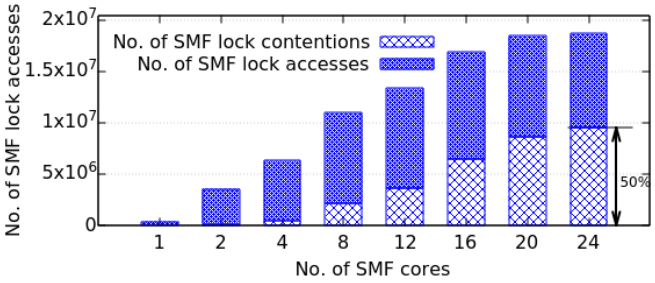



Fig. 1: Lock accesses and lock contentions in SMF

granularity of mobile subscribers. The NFs are typically built as multi-threaded software, and all NF threads access shared data structures containing mobile user state using locks to ensure serialization. At high number of CPU cores, NF threads may often need to wait to acquire locks due to high lock contention, leading to lower NF throughput. To assess the extent of this problem, we instrument one of the NFs in an optimized production-grade 5G core (SMF, see Section V) to measure the contention on its most contended application lock. Figure 1 shows the number of lock acquisitions in the NF during an experiment, and the number of these lock accesses where the application thread had to wait due to contention. We see from the figure that about half of the lock acquisitions saw contention when the NF is running at saturation throughput on 24 cores, though this fraction is much lower at 7% at 4 cores.

In this paper, we investigate the problem of improving multicore scalability of stateful NFs running over the linux kernel, which maintain state at a granularity that does not map to TCP/IP flows, e.g., NFs in the control plane of the 5G packet core, because previous work fails to address the issue of multicore scalability for such NFs. We begin with the observation that if the network stack can partition traffic to cores based on application keys (e.g., mobile subscriber identifiers) instead of the connection 4-tuple, we can ensure affinity of application keys to cores, i.e., we can have all traffic of a particular mobile subscriber land on the same core always. With such *application-aware packet steering* in place, NF threads can maintain state in per-core slices and access it without locking, leading to a lockfree, multicore-scalable design of the NF.

Note that there are other ways to perform application-aware steering without requiring network stack support. For example, we can steer requests of a specific key to a particular application thread in user space using a request dispatcher within the application. However, in such a design, the dispatcher thread may itself become the bottleneck, limiting the scalability of the NF. Therefore, this paper investigates the approach of application-aware packet steering performed within the network stack itself, which is a cleaner and more scalable approach. We also focus on compute-intensive NFs like those found in the 5G core, which perform significant computation on each request, and hence use locking extensively to access application state. The problem of eliminating application-layer locking is much less interesting in the case of network I/O-intensive applications like simple key-value stores. Finally, we note that there are many ways to enable lock-free operations within the application itself, including lock-free data structures. In this work, we limit our scope to achieving lock-free operation via application-aware steering of requests and sharding of application state across cores.

One of the main challenges in performing application-aware steering for NFs like the 5G core NFs is that the application-layer identifiers are embedded deep within the payload and not within standard TCP/IP headers. Tracking these identifiers is non-trivial because the size of the identifier and its position within the packet both vary based on the type of the message. Further, with persistent connections, requests from different users can be bundled together over the same connection, and sometimes even within the same network packet. Given these complications, the standard network stack mechanisms used to steer TCP connections to cores based on hash of the TCP/IP header fields do not suffice to perform application-aware steering in our use case. The key idea of AppSteer is to expose an API to let the sender embed application identifier into various transport layer header fields, e.g., TCP source port numbers, SCTP stream identifiers. AppSteer uses various techniques like eBPF (extended Berkeley Packet Filter) programs and other kernel-level packet steering mechanisms to steer traffic to CPU cores using these application identifiers, ensuring that the traffic of a given application identifier always reaches the same core, across all types of NFs.

With such a network stack capable of application-aware packet steering in place, we modify two important control plane NFs (AMF and SMF) of a production-grade 5G core implementation to work over AppSteer, by partitioning the internal application state into per-core slices and accessing it without any userspace locking. We find that our modified lockfree NFs have 15–18% higher saturation throughput as compared to their locking-based counterparts when running on 16 cores. These performance gains, accruing from significantly reduced contention for application locks—a measurement similar to the one shown in Figure 1 results in zero lock contentions in our prototype—are especially important for the performance-sensitive usecases of 5G that require high control plane throughput and low latency. Achieving higher throughput for the same amount of CPU resources will also translate to cost savings for mobile operators.

Our work makes the following contributions: (i) AppSteer enables application-aware packet steering to cores for multithreaded NFs, thereby enabling lockfree NF design for improved multicore scalability. While the idea of application-aware packet steering itself is not new, AppSteer's treatment of the idea is broader in scope than what has been considered in prior work [18], [19]. (ii) We provide a complete design and working implementation of AppSteer over the Linux network stack, considering all the complexities of packet processing in the Linux stack. (iii) We demonstrate the benefits of AppSteer by building lockfree versions of 5G packet core NFs running on top of AppSteer, and comparing them with their optimized locking versions.

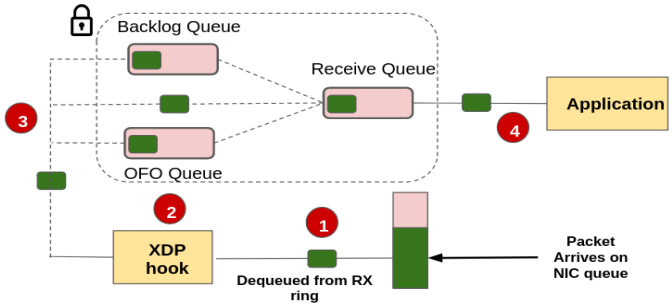



Fig. 2: Packet flow in the Linux network stack

## II. Background and Related Work

### A. Linux network stack

**Overview of packet flow.** Figure 2 shows the steps involved in the reception of a packet in the Linux network stack. ① Kernel network stack dequeues the packet descriptor from the RX queue of the network device, reads the packet from the DMA buffer, wraps the packet in the sk_buff structure (representation of a packet in the kernel), ② and before sending sk_buff to the upper layers for protocol processing, the sk_buff is handed off to an eBPF program (if installed by user application) running at the XDP hook. ③ For data packets received on established connections, the sk_buff is placed into some intermediate queues before finally reaching the socket receive queue to handle out-of-order packets. ④ Once an sk_buff is added to the receive queue of the destination socket after protocol processing, any application thread waiting on a read event on the socket (e.g., using the `read` or `epoll_wait` system calls) is woken up. When the application thread reads from the socket, the packet is dequeued from the socket receive queue and copied into the userspace buffer. Note that all operations on a socket, including the bottom half enqueuing a packet into the socket queue, or the application reading/sending a packet, are protected by a single socket lock.

**Listen sockets and SO_REUSEPORT.** Besides data packets arriving on established sockets, new connection requests (SYN packets) can also arrive for listen sockets. To enable multiple application threads to accept connections on the same server port without locking the shared listen socket, Linux supports the SO_REUSEPORT socket option [20]. This option allows us to have multiple listen sockets listening for new connections on the same transport layer port, and all sockets listening on the same port belong to a reuseport group. During the listen socket lookup procedure on receipt of a SYN packet, if the matched socket belongs to a reuseport group, then the kernel selects a socket out of that reuseport group randomly, ensuring that new connections are distributed as uniformly as possible between all listen sockets belonging to that reuseport group.

### B. 5G Mobile Packet core

The 5G packet core consists of several components that implement control plane and data plane functionalities for mobile subscribers. The Access and Mobility Function (AMF) is the entry point to all other control plane network functions in the packet core. The base station maintains a persistent SCTP connection with the AMF and exchanges messages over this persistent connection. A user that wishes to connect to a mobile network for the first time sends an initial registration request to the AMF via the RAN. The AMF communicates with other control plane NFs in order to authenticate the user, setup security keys and complete the registration procedure. All the NFs in the control plane communicate with each other over REST APIs, implemented over HTTP2 connections. After registration, the user establishes one or more sessions to transfer voice or data packets through the packet core. These session related messages are first received by the AMF, which then passes them on to the Session Management Function (SMF). Once the sessions are established, the UPF in the dataplane forwards user traffic according to the QoS parameters requested by the user.

Our inspection of the codebase of AMF and SMF from a production-grade 5G packet core implementation [21] shows that they are designed as multi-threaded applications listening for requests from a single server listen socket over a persistent connection, and/or from multiple listen sockets over short connections. An NF running on a N-core system has N threads, each pinned to a core, and each listening on a separate epoll instance for network events. AMF has an SCTP server socket and listens for SCTP association requests from RANs that wish to connect to it. Once connected, the RAN and AMF maintain a persistent SCTP connection for the duration of the association, and all messages coming from different UEs belonging to the RAN are received over the same SCTP connection at the AMF. The communication between AMF and SMF (and other control plane NFs) can happen over a single long running persistent TCP connection (where all request/responses are exchanged over this single TCP connection), or it can happen over multiple TCP connections (where each request/response is exchanged over a separate TCP connection). Both AMF and SMF maintain some context for every mobile subscriber (user) that they are handling, and read/write to this state during the processing of various signaling messages.

### C. Related Work

**Optimizations to the kernel network stacks.** Prior work suggests various optimizations to make the kernel network stack multicore scalable. Affinity-Accept [9] suggests having per-core listen sockets in the accept() system call, instead of all application threads accepting connections from a single listen socket. Megapipe [13] and Fastsocket [8] propose splitting various kernel data structures like the socket hash table into per-core slices. Such solutions benefit the case of short connections where there are connections to be accepted at scale (Affinity-Accept), and high number of entries in kernel data structures (Megapipe and Fastsocket). These solutions are orthogonal to our work as they cannot be used to perform application-aware packet steering inside the kernel, and hence cannot eliminate userspace locking.

**Kernel bypass network stack optimizations.** Many systems bypass the kernel to achieve high performance networking by using SR-IOV [22], or libraries such as DPDK [16] or

netmap [23]. Examples include mTCP [7], IX [11], Stackmap [14], and TAS [10]. These solutions rely on RSS [15] to split traffic to cores, and then use per-core TCP data structures to avoid lock contention. RSS uses the hash of the connection 4-tuple to steer incoming packets to various kernel and application level threads for network stack and application processing. Such solutions cannot steer packets to cores based on application-layer identifiers. Such solutions will not also work in the case of persistent connections, as all packets of a flow (possibly belonging to different users) will consistently be steered to a single CPU core. Also, these solutions require bypassing the kernel which may be hard to do in the cloud.

**Kernel bypass request scheduling.** ZygOS [12], minOS [24] and Shinjuku [25] schedule incoming requests to cores in order to improve tail latencies, by scheduling requests to idle cores. Shenango [26] combines scheduling with dynamic scaling to provide high CPU utilization. Such solutions schedule requests to run on various cores based on the workload the application is subjected to, and not based on any application layer identifiers. Limitations with kernel bypass network stack optimizations discussed above also apply here.

**Application-aware request steering.** While prior work proposes techniques to perform application-aware packet steering, none of them works across all NFs like those found in the 5G core, as described below. Mica [18] is a high-performance key-value store that aims to do application-aware request steering based on application keys, to avoid locks during write operations. Mica embeds application identifier in the source port of the request and uses that to steer requests at the receiver. However, Mica only works in the case of UDP and uses kernel bypass mechanisms and RSS to perform the steering, hence suffering from similar drawbacks discussed above. Syrup [19], a recent framework for application-defined CPU scheduling and packet steering inside the linux kernel, is the closest to our work. Syrup uses eBPF hooks at the SO_REUSEPORT logic, and can be programmed easily to perform application-aware packet steering. However, Syrup redirects all packets of a flow to the same receive socket, and hence cannot be used to perform application-aware packet steering when requests of multiple users come over a long-running transport layer connection.

## III. Design

### A. Design Goals

Most previous work (§II-C) deals with multicore scalability of simple NFs like NATs and load balancers that maintain TCP flow state and rewrite packet headers using this state. However, there are many NFs, like those in the 5G packet core, that do not fit this mold, for the following reasons. (i) They are transport layer endpoints (TCP/SCTP), and not simply packet-header rewriting NFs like NATs. This means that the incoming request would have to go through transport layer protocol parsing and processing before being handed to the NF. (ii) They maintain state at a granularity that is application-specific (e.g., mobile subscriber), and not at the granularity of TCP flows. (iii) There is no one-to-one mapping between

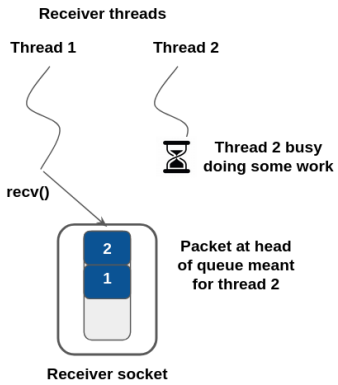



Fig. 3: Head of Line (HOL) blocking

application-layer entities and transport layer connections. That is, requests of a given application-layer entity can arrive over multiple transport layer connections. Conversely, one transport layer connection can see traffic pertaining to different application-layer entities. The primary goal of AppSteer is to redesign the network stack to enable lockfree operation of such NFs via application-aware packet steering, i.e., by redirecting traffic of a particular application-layer entity to the same core always. A secondary goal is to provide an API for enabling application-aware steering to NFs running over the Linux kernel, so that the solution can be widely used across a variety of NFs.

### B. Design Challenges and Key Ideas

**Embedding application-layer identifiers in packet headers.** The goal of AppSteer is to redirect every application-layer request to the same application thread (or core, assuming threads are pinned to cores) always, so that state can be maintained in per-core slices and accessed without locking. Performing this application-aware steering requires some agreement between the sender and receiver. Senders embed application-layer identifiers (app id) to outgoing packets via the API exposed by AppSteer. The receiver installs rules inside the kernel, mapping the complete app id space among different NF threads (each having a unique id, which we will refer to thread id from now onwards). Embedding the app id in the transport layer protocol headers lets us extract the app id from each incoming request at the receiver without parsing complex application data.

**Head of line (HOL) blocking and per-thread queues.** In the context of application aware request steering, when multiple application threads read data from the head of a socket queue, but specific threads can only process specific requests for which they store state, there will be head of line blocking as shown in Figure 3. For example, suppose thread 1 performs a recv system call on a socket, but the request at the head of the queue must be steered to another thread 2 that is currently busy and cannot receive. Thread 1 is blocked from making progress in this case. AppSteer overcomes this challenge by introducing per-thread queues, enabling each NF thread to read incoming requests from its own exclusive queue, thereby removing any HOL blocking. Note the analogous treatment of established sockets and listen sockets in our solution. In the case of requests arriving over

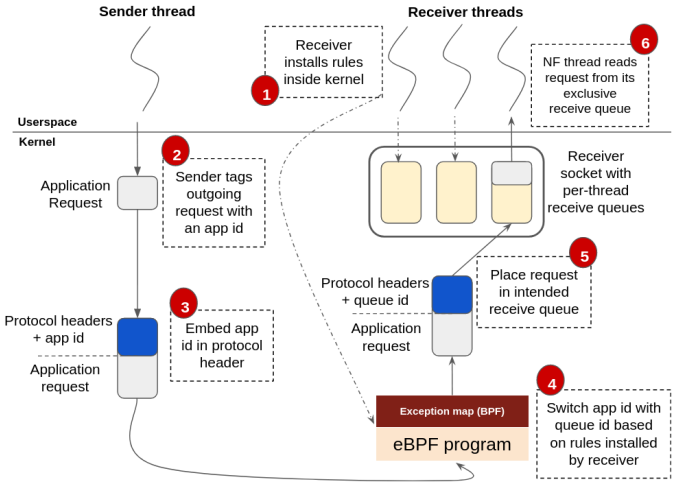



Fig. 4: Design Overview

a single long running persistent transport layer connection on an established socket, AppSteer provisions per-thread socket receive queues via careful changes to the Linux kernel, and redirects requests of a thread to its corresponding queue. In the case of short connections, where each request arrives over a new connection, we provision per-thread listen sockets (and in turn, per-thread accept queues) listening on the same port via SO_REUSEPORT, and redirect new connection requests to the suitable listen socket/accept queue.

**Redirecting requests to queues using eBPF.** We have so far described two key ideas of our solution: we embed app ids in the packet headers of incoming traffic, and we provision per-thread queues so that each thread can receive traffic destined to it in its own queue without HOL blocking. Now, how do we redirect traffic of a particular app id to its correct thread/queue? Note that RSS, which splits kernel level packet processing of incoming requests among different cores using the hash of the TCP 5-tuple, is not enough, because RSS has no control over which receive queue (or listen socket) incoming requests are appended to. Instead, AppSteer uses an eBPF program which parses the transport layer protocol header of incoming requests to replace the app id corresponding to the request with the correct queue or socket identifier, based on the mappings installed by the receiver NF. We use this embedded queue id (or socket id) to deliver requests to the intended receive queue (or listen socket), and therefore to the intended NF thread, via careful changes to the packet processing logic in Linux.

### C. Design Overview

Figure 4 illustrates our design. ① Receiver NF has per-thread receive queues (or listen sockets) wherein each thread of the NF reads requests from an exclusive receive queue (or listen socket). Receiver NF uses APIs provided by AppSteer to install rules mapping each NF thread to a particular per-thread receive queue or listen socket, and mapping the app id to receive queue id (or listen socket id). ② The sender tags outgoing traffic with an app id, again using an API provided by AppSteer. ③ AppSteer embeds this app id inside the transport layer protocol header before sending it to the receiver. ④ AppSteer uses the provided mappings in an eBPF program to switch app id with receive queue id (or listen socket id). ⑤ This queue id (embedded in the same part of transport layer protocol header as app id) is used further up the network stack at the receiver to deliver an incoming request to the intended receive queue (or listen socket), and ⑥ in turn to the intended NF thread because of a one-to-one mapping between NF threads and receive queues (or listen sockets).

AppSteer design described so far expects the receiver NF to statically shard NF state among different NF threads. This could lead to a case of load imbalance at the NF if a particular partition of an NF thread receives disproportionately higher load. This would leave other NF threads underutilised and a single thread overloaded. To overcome this issue we present the receiver NF with an exception map in which it could install rules to redirect certain requests belonging to an app id to a receive queue (or listen socket) different from the one mentioned in the static rules installed before hand. The receiver can monitor its load on the various threads and populate the entries in this exception map at runtime.

### D. Sender Design

AppSteer exposes a modified send() API to the sender NF which allows it to tag each outgoing request with an app id. We embed a hash of this app id inside the transport layer protocol header associated with the outgoing request. In the case of TCP, which is a byte oriented protocol, the kernel might coalesce multiple requests under a single TCP header before sending it out on the network. Because we need each outgoing request to be sent out with its own TCP header, containing its own hash of app id, AppSteer stops the kernel from trying to coalesce multiple requests encapsulated by a single TCP header at the sender via changes to the kernel sender logic at several places. We also switch off segmentation offloads at the sender for the same reason. Because we are primarily focusing on compute-intensive NFs which do not have a network I/O bottleneck, these changes are not expected to negatively impact NF performance. Note that we do not need to stop coalescing of requests in the case of SCTP, which is a message oriented protocol, because even if multiple requests are coalesced, each request gets its own SCTP header. This enables independent steering of even coalesced requests.

### E. TCP Receiver Design

The TCP receiver processing in AppSteer proceeds as follows. For receivers acting as servers for long-running TCP connections, we provision per-thread receive queues in the receive socket of established connections. For receivers acting as servers of short connections, we provision per-thread accept queues and listen sockets via SO_REUSEPORT. Each of these per-thread receive/accept queues is assigned a unique identifier. Every packet received from an AppSteer-compliant sender will contain the app id in the transport layer headers. AppSteer allows the receiver NF to provide mappings to map the app id space to the per-thread queue identifiers for the TCP sockets at the receiver. This mapping is used in an eBPF program at the XDP hook (for long running connections) or the listen socket lookup module (for short connections), to switch

the app id on incoming requests with the queue id. This queue id is used by the kernel further up the stack when adding an sk_buff to a per-thread receive queue. The recv() system call is changed to read requests from the head of the per-thread receive queue mapped to the NF thread that is making the system call, in order to avoid HOL blocking. Considering that most NFs make use of event driven programming, AppSteer modifies the epoll subsytem to be aware of our design. The epoll_wait() system call is programmed to return an EPOLLIN event only if the per-thread receive queue mapped to the NF thread making the epoll_wait() call has an outstanding request.

Much like at the sender, AppSteer intervenes at all of the places in the receive datapath where an sk_buff is being added to a queue or is moved from one queue to another, and stops coalescing of incoming sk_buff with the tail of receiving queue. We also switch off GRO to further stop coalescing of sk_buffs at the start of packet processing.

One challenge with having per-thread receive queues in AppSteer is that it enables the NF threads to read data received at the socket out of order, possibly violating TCP's in-order delivery state machine that assumes a single receiver queue. While this may not matter much to the application itself (which must expect and handle out of order processing of requests in a multithreaded application), this out of order processing of received data will break TCP state management. Therefore, AppSteer decouples the transferring of requests to the NF for processing, and the TCP state management which includes updating the receive window, sending out acks etc. These two stages are tightly coupled inside the vanilla kernel, wherein a recv() system call triggers packet copy of the request to the NF userspace buffer, and simultaneously updates all state pertaining to the TCP connection that is affected by recv(). To achieve said decoupling, AppSteer adds the incoming request not just in the per-thread receive queue, but also in the original receive queue in the TCP socket. When the recv() system call reads a request from a particular per-thread receive queue, it also marks that particular request as read in the original receive queue. The original receive queue still maintains all requests in-order, and AppSteer uses this queue for TCP state management. As soon as the request at the head of the original receive queue is marked read, AppSteer starts clearing requests until it encounters a request marked unread. AppSteer also invokes the TCP state management machine to move the TCP receive window and send out acks as it keeps clearing requests from the original queue. This gives the TCP state management logic the illusion that all requests are being read in-order, and preserves the correctness of the TCP state machine.

### *F. SCTP Receiver Design*

The handling of SCTP transport layer processing in AppSteer is similar to the TCP processing to a large extent. Similar to TCP, we have per-thread receive queues, and we add requests to a particular receive queue based on the app id attached by the sender in the SCTP header. Similar to that of TCP, we program the recv() system call to succeed and the epoll_wait() system call to return an EPOLLIN event only if there are requests in the receive queue mapped to the NF thread making the recv() and epoll_wait() call. However, in the case of TCP, the XDP program that switched app id with queue id is relatively simple because AppSteer ensures that each packet only contains a single request and hence we have to parse a single TCP header to make the switch. But, in the case of SCTP, there may be multiple requests present inside a single SCTP packet, albeit with separate SCTP headers. Each of these requests are called data chunks. There might also be control information related chunks that do not carry any SCTP data. The XDP program used by AppSteer carefully parses each SCTP data chunk header, switching app id with queue id, and skipping any control chunks in the packet.

Every SCTP data chunk also contains a stream sequence number (SSN), and SCTP expects chunks to be delivered in the order of their SSN. This expectation of ordered delivery can be switched off by setting the unordered bit in the flags of the data chunk header, in which case the SSN is completely ignored by the SCTP protocol layer. This is also something that the XDP program does while switching app id with queue id. This simplifies our design further up the stack, because there being no need for ordered delivery, and the transfer of data to the user application and the SCTP state management relating to that transfer are no longer coupled. As a result, AppSteer only adds an incoming request to the per-thread receive queue and not the original receive queue, and the recv() system call can handle the SCTP state management as well.

## IV. Implementation

### *A. Kernel changes*

We modify the linux kernel (based on version 5.11) to implement AppSteer as described in §III. Our implementation adds  350 LOC to the Linux kernel. We modify the socket structure to accommodate multiple receive queues, and add code to enqueue an incoming packet to the correct receive queue based upon the rules installed by the NF. We also the epoll subsystem of the linux kernel to notify only the thread running on the core for which the received packet is destined to. Finally we program the recv system call to read packets from a specific receive queue based on the rules installed by the NF. AppSteer makes use of the stream id field inside the SCTP header to hold app id. The stream id field length is 16 bits, and hence the app id keyspace in our current implementation 2^16 in the case of SCTP. In the case of TCP, we use the reserved field inside the TCP header to hold the app id. This field is 4 bits wide and hence the app id keyspace is only 16 in our current implementation of the TCP design. Although this limits the number of cores that we can scale our NF to 16 in our current implementation, the implementation could easily be made more scalable by using a different field (such as a TCP option field) inside the TCP header that holds a longer app id, and we leave this exploration to future work.

### *B. API exposed to NFs*

`send(request, app_id)` This API is used by the sender to tag each outgoing request with an app_id. When the request

is to be sent over persistent connections, AppSteer passes the app_id along with the modified write() system call, and then embeds the app_id in the protocol header inside the kernel. In the case of short connections, AppSteer creates a source port number with the app_id embedded in it, and binds the sender socket to that port before establishing a connection and sending a request over that connection.

`map_thread_to_queue(queueid)` This API is used at the receiver to map the calling NF thread to a per-thread receive queue (or listen socket). AppSteer does not allow multiple threads to be mapped to the same queue.

`map_appid_to_queue(app_id)` This API is used at the receiver to divide the entire appid keyspace among the per-thread receive queues (or per-thread listen sockets in the case of short connections). The receiver NF implements this API, by adding logic to return a queue_id based on the app_id passed to the API. AppSteer calls the implemented API from our eBPF programs (at XDP hook or at socket lookup module) to map incoming requests containing app_id to a particular receive queue (or listen socket) identifier.

`add_exception(app_id, queue_id)` This API is used at the receiver to add exceptions to the steering logic of the NF. AppSteer adds the app_id to queue_id mappings in an eBPF exception map, which is used by the eBPF programs in AppSteer to change the steering if needed (§III-C).

### C. Example NFs: AMF and SMF in 5G core

To test the benefits of performing application-aware steering with AppSteer, we port NFs from a production-grade 5G packet core implementation [21] to run over AppSteer. We describe our implementation of AMF and SMF because these are the most complex NFs in the 5G packet core. Both NFs maintain state at the granularity of the mobile subscriber, in various maps which have various user identifiers as keys. This state is shared across the multiple threads of the NF, which are pinned to a core and run an event-driven epoll loop. The NFs access this state with suitable locking to handle various signaling messages from mobile users, e.g., to register a user, or to setup a data session for the user. These NFs also communicate over both long-running persistent connections and short connections over TCP or SCTP with other NFs. We compare the performance of these baseline NFs with the lockfree versions built over AppSteer.

To build lockfree versions of these NFs over AppSteer, we modify both NFs to maintain state in per-core slices that can be accessed without locking. In order to perform application-aware packet steering, we embed the hash of user identifiers in the protocol headers using our API at the sender. At the receiver, we install rules using our API to deliver a request to the desired queue id. We also assume that the number of cores assigned to the NF does not change dynamically, so that the mappings are stable.

## V. Evaluation

### A. Experiment Setup

**Load Generator.** The load generator in our implementation is a RAN emulator software that emulates a specified number of concurrent users (each user denoted by a thread inside the RAN emulator), and generates signaling load from these users in a closed loop manner towards our packet core NFs. We use the initial registration and session establishment procedures to measure the performance of the AMF and SMF respectively, because these procedures consume significant computation resources at the corresponding NFs.

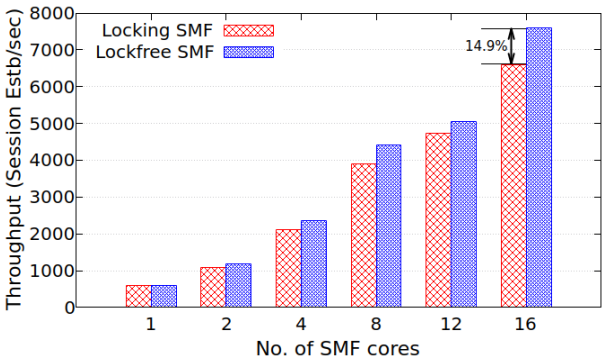



Fig. 5: SMF Throughput (persistent TCP)

| Number of cores | 1 | 4 | 8 | 16 |
|---|---|---|---|---|
| Rate of epoll events in locking SMF | 1214 | 4599 | 7882 | 13347 |
| Rate of epoll events in lockfree SMF | 846 | 3314 | 5989 | 9496 |

TABLE I: Rate of socket events at SMF (per second)

**Experimental setup.** We host both the locking and lockfree NFs on two commodity multicore servers running Ubuntu 21.04 with Linux kernel version 5.11 (with our kernel-level changes incorporated when running with lockfree NFs). Both machines are connected directly without any switches in between. One of the servers hosts the NFs which generates traffic towards our NF under test, while the other server hosts the NF under test. In the case of 5G packet core, this server can host other NFs which the NF under test exchanges messages with to complete a particular callflow of a 5G core signaling procedure. All NFs hosted on the same machine are assigned mutually disjoint sets of cores in order to not impact each other's performance.

**Parameters and metrics.** We vary the number of threads in the client to increase the load on the NF under test (AMF, SMF or echoserver), and measure its performance when it reaches its maximum capacity at peak CPU saturation. We measure this saturation throughput (control plane procedures/second) when different number of cores are assigned to the NF under test, thereby measuring its multicore scalability, i.e., how its capacity scales with increasing cores.

### B. Evaluation of SMF

We compare the performance of our baseline locking-based SMF (running on vanilla kernel) with lockfree SMF running on top of AppSteer, when the SMF receives session establishment requests over a persistent TCP connection, i.e., all session establishment requests come over the same TCP connection. Figure 5 plots saturation throughput (number of session establishments completed/sec) of both designs as a

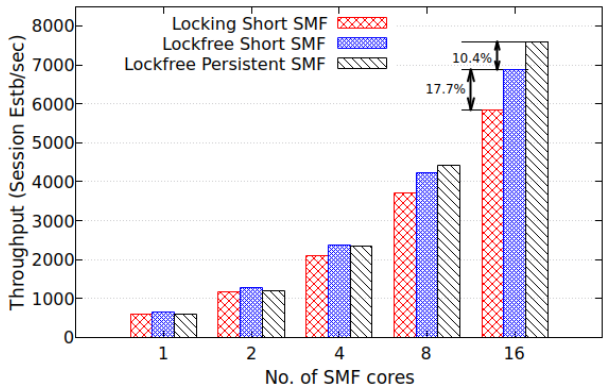



Fig. 6: SMF Throughput (non-persistent/short TCP)

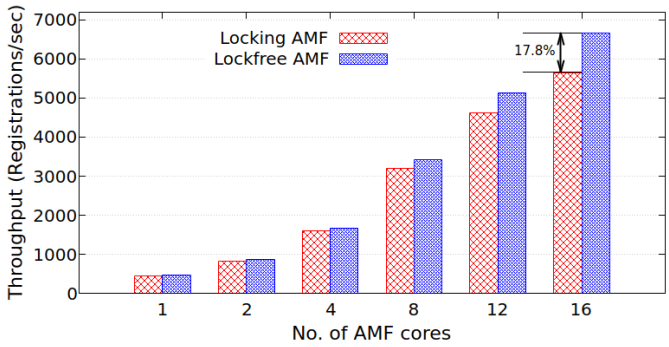



Fig. 7: AMF Throughput (persistent SCTP)

function of the number of cores SMF runs on. We see an improvement of 14.9% in throughput at 16 cores in the case of lockfree SMF when compared with locking SMF. Most of these gains come from eliminating locks to access state stored at SMF in the lockfree SMF design made possible by AppSteer. Figure 1 plots the ratio of lock accesses which end up being contended at SMF in the baseline locking-based SMF, which we completely eliminate in the lockfree SMF design. Some part of these gains also come from optimizing notifications to threads in the epoll subsystem of AppSteer. When a multithreaded NF runs on the vanilla Linux kernel and waits for events in an epoll wait loop, all threads are woken up when a request arrives on a socket. With the use of per-thread receive queues, only the thread responsible for servicing the request is woken up in the lockfree NFs running on AppSteer. Table I shows the rate of socket events per second both versions of SMF encounter with varying number of cores. We experience a 28% drop in the rate of socket events in the case of our lockfree design when compared with the locking design. This makes AppSteer more CPU efficient as we avoid spurious wakeups, which in turn contributes to the gains.

Next, we compare the performance of our baseline locking-based SMF (running on vanilla kernel) with lockfree SMF running on top of AppSteer, when the SMF receives session establishment requests over separate short connections, i.e., each session establishment request comes over a new TCP connection. Figure 6 shows the saturation throughput (session establishments/second) as a function of the number of SMF cores for both designs, and also compares this throughput against the saturation throughput in the case of lockfree SMF receiving requests over a persistent TCP connection. We see from this figure that our lockfree design scales much better than the locking-based baseline over short connections, and has a 17.7% higher saturation throughput at 16 cores. Further, as expected, the lockfree SMF that uses persistent connections performs better than the lockfree SMF that uses short connections because it avoids TCP connection establishment and teardown overheads. This result highlights the benefits of AppSteer over prior work like Syrup [19] which can be used to perform application-aware request steering only over short connections and not for persistent connections.

### C. Evaluation of AMF

| Number of Cores | 1 | 4 | 8 | 12 | 16 |
|---|---|---|---|---|---|
| Contention ratio(in %) | 0 | 10.1 | 27.4 | 44.4 | 55.8 |

TABLE II: AMF Lock Contention

Next, we compare the performance of our baseline locking-based AMF with the lockfree AMF built over AppSteer. In this experiment, the AMF receives user registration requests from the RAN over a long running SCTP connection. Figure 7 plots the saturation throughput (number of users registered/sec) of both designs as a function of the number of cores AMF runs on. We see an improvement of 17.8% in throughput at 16 cores in the case of lockfree AMF when compared with locking AMF. Most of these gains come from eliminating locks to access state stored at AMF in the lockfree AMF design. Table II shows the ratio of lock accesses which end up being contended at AMF in the baseline locking-based AMF, which we completely eliminate in the lockfree AMF design by running on top of AppSteer. Much like in the case of SMF, we also see 10% fewer socket events in the case of our lockfree design when compared with the locking design, which also contributes to the gains seen with AppSteer.

### D. Overhead analysis

**Kernel processing overheads.** AppSteer makes changes to the Linux kernel as described in section §III to achieve application-aware packet steering in the case of TCP and SCTP sockets. These changes result in some overheads being added to the packet processing pipeline of the vanilla kernel, especially when processing packets over long-running persistent TCP and SCTP connections, where we add per-thread receive queues and perform extra work on them. We define the average hold time of a lock as the average time which a thread holds the lock for after acquiring the lock. Since all operations on a socket are protected by a single lock (see §II), the average hold time of a socket lock approximates the processing time per socket operation, and a comparison between the average hold times of the kernel socket lock in our locking and lockfree designs will give a good indication of the overheads we add to the packet processing pipeline in order to perform application-aware steering. Table III shows the average hold time of the socket lock for SMF (running on persistent TCP) and AMF (on SCTP), for both the locking and lockfree designs at 16

cores, obtained from `/proc/lock_stat`. We see from this table that our design does not add much overhead to the SCTP packet processing, as we do not need to make many changes to the SCTP state management. However, we add about 0.5 us extra overhead to the TCP packet processing pipeline because we have to handle both the original TCP receive queue along with the per-thread receive queues, which increases the socket lock hold time significantly. However, this increase in overhead in the kernel packet processing of AppSteer is more than compensated by the elimination of userspace locking with AppSteer, resulting in a net increase in performance when running lockfree NFs over AppSteer. Therefore, AppSteer is beneficial mainly in NFs that are compute intensive and perform significant computation over shared state while holding locks in userspace, where the benefits of lockfree operation outweigh the costs imposed by AppSteer in the form of extra overheads in the kernel. It is important to note that AppSteer does not add kernel processing overheads in the case of packets received over short connections, because we use the kernel's SO_REUSEPORT to provision per-core accept queues, and we simply replace the socket matching logic of the kernel in the case of SO_REUSEPORT with our own eBPF program to facilitate application-aware steering.

**eBPF overheads.** Next, we measure the overhead incurred by our eBPF programs when rewriting packet headers to map app id to queue id for application-aware request steering. Because the eBPF processing in the case of SCTP is more complex than that at TCP, we measure the overhead of the eBPF program used to steer requests over SCTP at AMF. Each SCTP packet being parsed might have multiple chunks (user requests), and our eBPF program parses each chunk to rewrite stream ids. Table IV shows the eBPF packet processing overheads as a function of the number of chunks in the SCTP packet. Note that 20 chunks is the maximum number of chunks that can be accommodated in a packet of length 1 MTU in our use case. We see from the table that the overheads are very low, with parsing 20 chunks taking only 23.6 microseconds. This is negligible overhead when the overall registration procedure at AMF itself takes order of milliseconds to process. Once again, the overheads added by AppSteer are compensated by the benefits accrued due to eliminating userspace locking, especially in NFs that are compute intensive and hold userspace locks to access shared state during such computation.

| | SMF (TCP) | AMF (SCTP) |
|---|---|---|
| Locking NF | 0.32 | 1.40 |
| Lockfree NF | 0.81 | 1.44 |

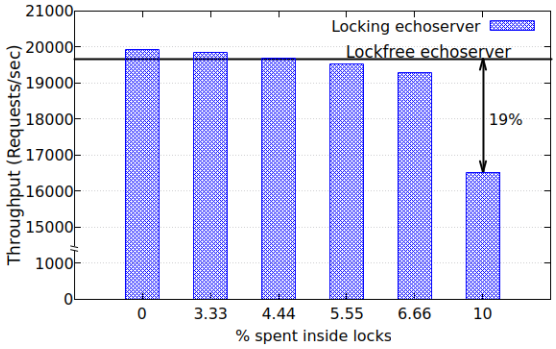
TABLE III: Avg. hold times of socket lock (in us)

| Number of chunks in the SCTP packet | 1 | 4 | 8 | 16 | 20 |
|---|---|---|---|---|---|
| Time taken to parse the SCTP packet (in us) | 3.7 | 6.7 | 11.4 | 19.6 | 23.6 |

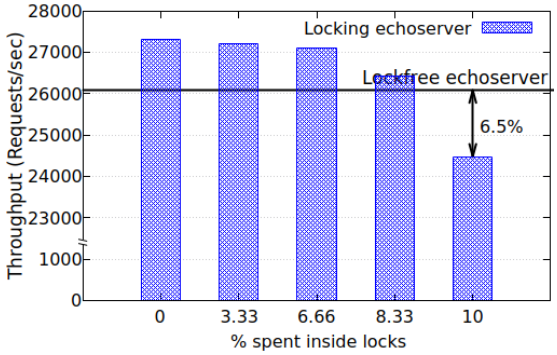
TABLE IV: Overhead of XDP program

### *E. Microbenchmarks*

So, which type of NFs will benefit from AppSteer and at what level of application-layer locking will we start to see the benefits? We perform some microbenchmarks to answer this question. We build a small echoserver to run on top of AppSteer, which receives a request, does some dummy work, and returns a response back to the client. The client embeds an app id in each of the request, and the receiver also installs some rules to evenly distribute load among all threads of the echoserver. We build two versions of the echo servers, one compute-intensive and one I/O-intensive, that differ in the amount of computation they perform on the dummy requests they receive. Both versions have locking variants that run on the vanilla kernel, and lockfree versions built with AppSteer API running on our modified kernel. The locking variant spends a configurable fraction of its time holding a lock while servicing the dummy request. Figure 8 shows the saturation throughput (at 8 CPU cores) of the locking echoserver version in both the compute and I/O intensive configurations as we increase the amount of locking in the echoserver. For comparison, we also show the saturation throughput of the lockfree echoserver (on AppSteer) in both configurations. Note that the throughput of the lockfree version does not depend on the amount of locking being varied on the x-axis. We see that in the case of the compute intensive configuration, the lockfree echoserver running on top of AppSteer does better than the locking echoserver if the locking echoserver spent more than 4.33% of its service time holding a lock. This number increases to around 8.33% and 10% in the case of I/O intensive




(a) CPU intesive echoserver




(b) IO intensive echoserver

Fig. 8: Throughput comparison between lockfree and locking echoserver designs under two different configurations of service time

configuration. This is because the CPU intensive echoserver saturates at a much lower packet rate when compared to the I/O intensive echoserver, and hence has lesser overhead added by AppSteer to the kernel packet processing. On the other hand, in the I/O intensive echoserver, packet rates are higher, which puts more load on the packet processing pipeline in the kernel, translating to higher contentions on the socket lock. Therefore, a higher locking percentage is required to offset the overheads of AppSteer inside the kernel to see any performance benefits. This result once again reinforces the fact that AppSteer is more useful for applications that are compute intensive and when the said computation requires holding application-layer locks.

## VI. Conclusion

AppSteer considers the problem of improving the multicore scalability of software network functions via application-aware packet steering. Prior work has dealt extensively with the problem of eliminating lock contention within the network stack by splitting kernel data structures into per-core slices, and using RSS to maintain flow level affinity to a core. These network stacks were mostly developed on top of kernel bypass mechanisms like DPDK, and they lead to multicore scalability of the NF only if it maintains state at the granularity of a transport layer flow. AppSteer extends the scope of this problem by considering NFs running on top of the linux kernel, and NFs which store state at the granularity of an application key which does not map to a transport layer flow. The AppSteer framework exposes APIs that let an NF steer network traffic to application threads based on application identifiers in the packets, and implements these APIs over the Linux kernel via eBPF programs and kernel changes. With such a mechanism in place, NFs can partition their state into per-thread slices and access it without locking, leading to multicore scalability. We use the NFs in the 5G control plane as a use case to demonstrate the benefits of AppSteer, because these compute-intensive NFs maintain state at the granularity of a mobile subscriber, access the state frequently to perform significant computation on each user request, and cannot be ported to run on kernel bypass network stacks easily. We modify the NFs of a production-grade 5G packet core implementation to maintain per-core state and operate in a lockfree manner running on top of AppSteer. We then compare the saturation throughput of our lockfree NFs with their optimised locking-based baselines running on vanilla kernel, and find that the lockfree NFs have up to 15–18% higher throughput on 16 cores. We also evaluate the costs and benefits of AppSteer and come up with clear guidelines on when application-aware steering is beneficial.

## References


[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[2] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. T. Morris, N. Zeldovich, *et al.*, "An analysis of linux scalability to many cores.," in *OSDI*, 2010.

[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, "Scaling memcache at facebook," in *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013.

[4] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The linux scheduler: A decade of wasted cores," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[5] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[6] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys (CSUR)*, 2012.

[7] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "Mtcp: A highly scalable user-level tcp stack for multicore systems," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[8] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi, "Scalable kernel tcp design and implementation for short-lived connections," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, 2016.

[9] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving network connection locality on multicore systems," in *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, 2012.

[10] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson, "Tas: Tcp acceleration as an os service," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[11] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, 2014.

[12] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017.

[13] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "Megapipe: A new programming interface for scalable network i/o," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*, 2012.

[14] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "Stackmap: Low-latency networking with the OS stack and dedicated nics," in *2016 USENIX Annual Technical Conference (USENIX ATC '16)*, 2016.

[15] T. Herbert and W. de Bruijn, "Rss," 2010.

[16] Intel, "Intel data plane development kit," 2018.

[17] 3GPP, "Mobility management entity (mme) sgs interface specification," 2020.

[18] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[19] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis, "Syrup: User-defined scheduling across the stack," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, 2021.

[20] M. Kerrisk, "The so_reuseport socket option," 2013.

[21] "Proprietary 5g core implementation.," 2021.

[22] "Sr-iov," 2018.

[23] L. Rizzo, "Netmap: A novel framework for fast packet i/o," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC '12)*, 2012.

[24] D. Didona and W. Zwaenepoel, "Size-aware sharding for improving tail latencies in in-memory key-value stores," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[25] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for microsecond-scale tail latency," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, 2019.

[26] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI '19)*, 2019.