eTran: Extensible Kernel Transport with eBPF

Zhongjie Chen[†] Qingkai Meng[‡] ChonLam Lao[¶] Yifan Liu[†]
Fengyuan Ren[†] Minlan Yu[¶] Yang Zhou[§]
[†]*Tsinghua University* [‡]*Nanjing University* [¶]*Harvard University* [§]*UC Berkeley & UC Davis*

Abstract

Evolving datacenters with diverse application demands are driving network transport designs. However, few have successfully landed in the widely-used kernel networking stack to benefit broader users, and they take multiple years. We present eTran, a system that makes kernel transport extensible to implement and customize diverse transport designs agilely. To achieve this, eTran leverages and extends eBPF-based techniques to customize the kernel to support complex transport functionalities safely. Meanwhile, eTran carefully absorbs user-space transport techniques for performance gain without sacrificing robust protection. We implement TCP (with DCTCP congestion control) and Homa under eTran, and achieve up to 4.8×/1.8× higher throughput with 3.7×/7.5× lower latency compared to existing kernel implementation.

1 Introduction

Datacenter applications are always evolving and becoming increasingly diverse. For example, recent microservice applications heavily rely on Remote Procedure Calls (RPCs) to communicate among loosely-connected components; storage applications require bulk data transfer and replications for fault tolerance; popular ML applications employ collective communication primitives such as AllReduce to exchange large volumes of model weights and gradients.

To better support these diverse applications, datacenter researchers have proposed many network transport designs. For example, the Homa transport [50, 55] with SRPT (Shortest Remaining Processing Time) and receiver-driven designs target low-latency RPCs; The DCTCP congestion control (CC) protocol [2] enhances the TCP transport for various datacenter workloads, including bulk transfers; The MLT transport [74] is designed specifically for distributed DNN training. There are many more transport designs for emerging datacenter workloads [14, 26, 30, 44, 45, 49, 51, 82].

However, very few transport designs have landed in the widely-used Linux networking stack to benefit broader users. For example, DCTCP took 4 years to land [18], while MPTCP took nearly a decade [51]; Homa (2018) [50] has been an author-maintained kernel module since 2021 but remains unmerged in mainline. To the best of our knowledge, no transport protocol has landed on the kernel since DCTCP. The key reason for the above status quo is that kernel transports take extremely heavy efforts to develop and maintain, and often give unsatisfactory performance. As a result, it cannot keep up with the fast evolvement and high performance requirements of datacenter transports.

People nowadays mainly resort to kernel-bypass techniques by implementing transport directly in the user space or in hardware. However, this approach largely sacrifices kernel-provided protection and manageability [4, 57, 61, 80, 81], not friendly to public cloud users. For instance, integrating transport within the application process [38, 78] risks malicious manipulation or unintended interference with transport events (e.g., acknowledgments, timeout). Alternatively, running transport in a separate process [41, 47] or on hardware [3, 82] introduces new infrastructure and tools, incurring high management burdens such as telemetry and debugging.

Instead, this paper takes the kernel approach and tries to tackle the following challenging problem: *how to make the kernel transport extensible to enable agile customization, while achieving kernel safety, strong protection, and high performance?* This is challenging because 1) customizability might expose new attack surfaces in the kernel, creating safety challenges; and 2) strong protection requires putting the entire transport states inside the kernel (e.g., the kernel TCP stack), which is usually contrary to high performance due to high kernel overhead (e.g., user-kernel crossing), and makes customization hard due to fixed kernel implementation.

Fortunately, we find that the growing trend of safe kernel extensions via the increasingly mature eBPF technique provides promising opportunities. This technique allows applications to inject statically-verified code into kernel eBPF hooks to change how the kernel reacts to certain events, e.g., incoming packets. Note that this is different from writing unsafe kernel modules that might crash the whole kernel. A bunch of recent eBPF verification work [8, 46, 69, 73, 76] has largely strengthened the safety of eBPF programs running inside the kernel. Recent advances in eBPF functionalities like XDP programmable network data path [58], dynamic memory allocation [20], exception handling [40], and exposing more kernel functions [42] make eBPF more and more capable.

We introduce eTran (**e**xtensible kernel **Tran**sport), a system for agilely customizing kernel transports. eTran achieves agile customizability and kernel safety by 1) leveraging existing eBPF infrastructure such as built-in data structures (eBPF maps), BPF timer, and XDP for fast packet IO, and 2) extending it with new eBPF hooks and maps to support complex transport functionalities while conforming to the strict eBPF verifier for safety. eTran allows safely offloading full trans-

port states and performance-critical operations into the kernel, achieving strong protection. For example, it supports packet acknowledging, flow pacing, fast retransmission, and more in eBPF. In terms of performance, eTran shows similarities with user-space transports for the following design choices: direct packet delivery to user-space without kernel-to-user copies; lightweight packet buffer management and syscall-free IO batching; streamlined transport implementation but preserving genericity. However, transport processing under eTran remains in the kernel, fully isolated from applications. Crucially, transport states are safeguarded by kernel eBPF, preventing direct access from untrusted libraries.

Under eTran, we implemented two representative and vastly different transports, the sender-driven TCP (with DCTCP congestion control) and the receiver-driven Homa, and show great performance gains. For TCP, eTran version achieves 2.4-4.8× higher throughput and 3.2-3.7× lower latency than kernel-native TCP, while being only 1.6-1.8× slower than a state-of-the-art microkernel-inspired kernel-bypass solution with comparable latency. For Homa, eTran version outperforms the kernel module implementation by 1.7-1.8× higher throughput and 3.9-7.5× lower latency for RPCs. We further showcase that eTran supports multi-tenancy, coexisting transport protocols and flexible traffic management.

2 Background and Motivation

2.1 Network Transports

Diverse transports for applications. Modern datacenters have developed various transports to better meet their workloads with two main categories as follows:

- *Sender-driven transports*, such as DCTCP [2], Swift [44], and more [1,45,49,72,75,82], heuristically control the sending rate on the sender side. These protocols are generally beneficial for elephant flows [32] or traffic patterns with low entropy. They are commonly used in machine learning training and storage applications [25,48,60].
- *Receiver-driven transports*, such as NDP [30], Homa [50, 55], and more [14, 26], assume last-hop congestion and proactively control the sending rate by allocating credits from receivers to senders [14,26,30,50]. These protocols prevent queuing and improve tail performance, especially for small flows that dominate RPC applications [15,50].

While datacenter transports differ in how to control the sending rate (e.g., different congestion signals), their implementation share the following common components:

- *Segmentation and reassembly* involve chunking data into packets for sending and reassembling them upon receiving.
- *Congestion control* explores the right sending rate to utilize the available network bandwidth while avoiding congestion. To adjust the rate, transports maintain various congestion states, such as RTT. To enforce rate control, both sender-driven and receiver-driven transports require pacing engines based on target rate, credit availability, or window size.
- *Loss recovery* detects packet loss and triggers retransmission. Transports usually leverage multiple ACK responses to infer packet loss and trigger fast retransmission. When this mechanism is insufficient, timeout-based retransmission serves as the last line of defense for reliable transfer.

Protection for transports. Transport implementation requires protection or isolation from the user-space applications, typically through kernel-based networking stacks. Here, we assume the following threat model that is common in public clouds: applications using transports might be malicious; the eBPF programs written by applications might be malicious but can be detected and prevented by the existing kernel verifier, recent advanced runtime verifier [40], or others (see §8); the kernel and eBPF subsystems are trusted, e.g., through verification techniques like [69]. Without protection, application bugs, crashes, or malicious behaviors could corrupt the transport states and impact other colocating applications [5]. Kernel-based protection has its additional benefit where the networking stacks can firewall malicious applications, enforce access control, and implement various rate-limiting policies [4,57]. On the other hand, eBPF, as a kernel-native mechanism, naturally provides kernel-based protection against user-space applications, which motivates us to explore the possibility of running transports inside eBPF.

2.2 eBPF Basics

eBPF (extended Berkeley Packet Filter) is a promising technique that allows users to run customized, verified programs at specific kernel hooks. It uses eBPF maps and eBPF helper functions to store and exchange various data types between kernel and user space. Overall, eBPF enables safe and efficient extensions of kernel capabilities without modifying kernel code or loading unverified (thus unsafe) kernel modules.

Increasingly capable eBPF. The Linux kernel community is advancing the eBPF subsystem to handle more complex logic. For instance, `kfunc` complements eBPF helper functions by exposing kernel functions and data structures with restrictions; `dynptr`, added in v5.19, allows pointing to dynamically-sized memory; a dynamic memory allocator was added in v6.1; and advanced structures like rbtree (Red-Black Tree) were added in v6.3. Although eBPF still has limitations (e.g., no floating-point arithmetic to avoid non-determinism [53]), its growing capabilities enable safe kernel transport customization.

Fast packet processing with eBPF. XDP (eXpress Data Path) [58] is an eBPF hook in the Linux kernel for fast packet processing; AF_XDP is a specialized socket family built on XDP for fast packet processing in the user space. Packets processed by XDP can be dropped, forwarded back, passed to kernel networking stacks, or redirected to AF_XDP sockets based on the return code specified by the eBPF program.

AF_XDP uses four rings for packet IO: TX/RX rings per socket and Fill/Comp rings per NIC queue. The application provides buffers via the Fill ring for the kernel to receive packets, which are then placed in the RX ring by the kernel.

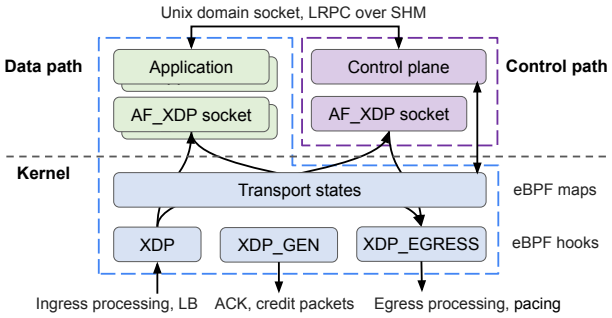


Figure 1: Overall architecture of eTran.

It submits buffers to the TX ring for transmission, and the kernel returns transmitted buffers to the Comp ring for reuse. All operations are asynchronous and optimized for syscall-free. Kernel operations of AF_XDP run in the NAPI context, which is Linux's event-driven packet processing mechanism. AF_XDP uses UMEM, a pre-allocated memory pool of fixed-size chunks, as the packet buffer pool. Currently, an AF_XDP socket can only bind to one NIC queue and UMEM while a NIC queue or UMEM can bind to multiple AF_XDP sockets.

3 eTran Design

eTran has the following four design goals:

- **Agile customizability**: eTran should allow agilely customizing diverse transports and their full functionalities.
- **Kernel safety**: eTran should prevent eBPF programs from tampering with or even crashing the kernel.
- **Strong protection**: eTran aims to maintain *all* transport states inside the kernel with corresponding state operations in eBPF, protecting against untrusted applications.
- **High performance**: eTran aims to achieve much higher performance than existing kernel transports and comparable performance to kernel-bypass ones.

Figure 1 shows the high-level architecture of eTran, which separates any transport into a control path and a data path. The control path handles non-performance-critical eBPF programs attaching/detaching, AF_XDP socket and UMEM creation/destroying, connection setup/teardown, and complex tasks that are unsuitable for eBPF (§3.1). It runs inside a root-privilege daemon, and multiple control paths from different transports can share the same daemon. Through this control path separation, eTran allows non-privilege applications to attach eBPF programs to the kernel through the control path daemon.

The data path handles performance-critical transport operations (e.g., packet reassembly, ACK/credit response, flow pacing) with streamlined implementation for high performance; it maintains all transport states (with state operations) inside the kernel for strong protection (§3.2). The data path runs across the kernel and user space: the kernel part runs in eBPF for kernel safety with direct access to transport states; the user-space part runs through untrusted libraries that cannot directly access transport states. Both parts can be agilely customized.

The data path and control path communicate through Unix domain socket and efficient LRPC [6, 24, 54] over shared memory. Additionally, eTran features flexible user-space IO by layering atop of the AF_XDP sockets (§3.3), multi-tenancy support, and flexible traffic management (§3.4).

3.1 eTran Control Path

Managing eBPF program. Applications submit transport-specific eBPF programs to our control path daemon to attach to the kernel, as such attaching requires root privilege. The daemon will first do a series of checks to make sure these programs submitted by different applications won't conflict with each other, e.g., overlapping ports or eBPF maps. Then, it will try to attach the eBPF programs to the hooks and create entries in `BPF_TYPE_PROG_ARRAY` for them, which will go through the kernel verification for liveness and safety.

Managing AF_XDP socket and UMEM. The control path daemon creates all AF_XDP sockets on behalf of applications and then transfers the file descriptors to applications through the Unix domain socket. This allows applications to use AF_XDP without privileges (e.g., `sudo`). Similarly, the daemon is also responsible for creating an individual UMEM for each application, mapping it into its address space. All sockets for a single application share the same UMEM.

Managing transport connections. The control path daemon handles connection management for connection-oriented transports such as TCP. Applications post control operations (i.e., `open/connect/listen/accept/close`) via LRPC channels; then the daemon intercepts the connection-related packets through its own AF_XDP socket and handles connection setup/teardown, then returns results to applications. eTran decides to put connection management into the control path due to the complexity of handshaking and timeouts, which are challenging for eBPF while being non-critical to transport performance. A similar design is adopted by TAS [41].

Managing CC and loss recovery. eTran decides to put few challenging-to-offload functionalities to the control path, including running advanced CC algorithms (e.g., involving floating-point arithmetic that is prohibited in eBPF) and handling severe loss recovery triggered by timeouts (not TCP fast retransmission). We will elaborate on this in §4.

3.2 eTran Data Path: Kernel Offloading

Running transport protocols and maintaining transport states inside the constrained eBPF environments requires enhancing the existing eBPF subsystem. We use the following three example issues to motivate the necessity of such enhancement:

① Many transports update states when packets get sent out, e.g., Homa decreasing the remaining bytes, but the XDP hook only works on ingress packets.

② Transports usually generate ACK and credit packets to coordinate senders and receivers, but the XDP hook does not support packet generation inside the kernel.

③ Almost all transports require a pacing mechanism to regulate when to send packets, but the XDP hook does not

support any buffering mechanism to defer packet sending.

The challenge is how to efficiently enhance the eBPF subsystem while not incurring destructive chances to eBPF, e.g., keeping similar semantics on the introduced changes, and reusing the existing kernel verifier and data structures. To address this challenge, eTran provides two new eBPF hooks, `XDP_EGRESS` and `XDP_GEN` (§3.2.1), and one new eBPF map, `BPF_MAP_TYPE_PKT_QUEUE` (§3.2.2).

3.2.1 New eBPF Hooks for Egress and Pktgen

XDP_EGRESS is a new eBPF hook that eTran designs to allow eBPF to efficiently customize egress packet processing, addressing the issue ①. eBPF programs attached to this hook accept an egress packet context as the input parameter and specify a return code to determine how this packet gets transmitted. We make the hook transparent to the NIC driver layer to avoid per-vendor code modifications. To achieve this, we place the hook in the vendor-agnostic AF_XDP kernel code, specifically, the `xsk_tx_peek_desc` function. This function fetches packet descriptors from AF_XDP TX rings and delivers them to the NIC drivers, finishing packet transmissions.

We also aim to reuse as many existing eBPF infrastructure as possible. First, `XDP_EGRESS` hook uses the same packet context (i.e., `struct xdp_md`) as XDP, facilitating the use of eBPF helpers designed for XDP. We add a new `umem_id` field at the end of the struct to indicate the UMEM of each packet, preventing unauthorized access to resources like connections and ports. For XDP programs, this field is invalid and inaccessible (see §4.1 for more detail). Second, the `XDP_EGRESS` hook shares the same return codes and actions as XDP:

- XDP_TX: Transmit the packet immediately.
- XDP_REDIRECT: Redirect the packet to a special map for buffering or another AF_XDP socket.
- XDP_DROP: Drop the packet.

XDP_GEN is another new eBPF hook that eTran designs to allow eBPF to efficiently generate packets. These packets can be customized based on the transport, e.g., ACK and credit packets, addressing the issue ②. eBPF programs attached to this hook also use the same packet context as XDP, and make packet-generating decisions based on the states from eBPF maps. Figure 2 shows how this hook works : first, the eBPF program at the XDP hook pushes necessary metadata (e.g., `seq` and `ack` in TCP) into a per-CPU eBPF queue, if it needs to generate an ACK packet; then the eBPF program at the `XDP_GEN` hook will pop any such metadata from the queue, fill it into a new packet frame, and send the packet out. eTran places the `XDP_GEN` hook in the `xdp_do_flush` function that is called at the end of a NAPI poll to generate packets.

A strawman approach dynamically allocates packet frames on-demand but incurs high overhead due to complicated and slow kernel memory management. Instead, we leverage batching to amortize the overhead: we pre-allocate enough frames and call the `XDP_GEN`-attached eBPF program in a finite loop whose budget is the same as NAPI; in each iteration, the eBPF

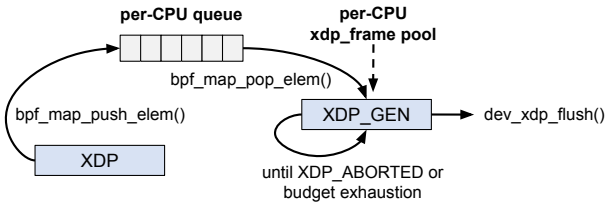


Figure 2: Coordination between XDP and XDP_GEN.

program may use a pre-allocated packet frame to generate a packet. We further leverage the *page_pool* allocator, an efficient memory allocator optimized for XDP, to do the frame pre-allocations. Doing batched packet generations essentially coalesces ACK or credit packets and sends them in batches for high performance; a similar coalescing mechanism has been leveraged in the transport implementations from Google Falcon [28], TAS [41], Caladan [24], and eRPC [38].

Our `XDP_GEN` hook supports three actions:

- XDP_TX: Transmit the ACK/credit packet immediately.
- XDP_DROP: Drop the ACK/credit packet.
- XDP_ABORTED: No more ACK/credit packets are generated, and the finite loop will be terminated.

3.2.2 New eBPF Map for Pacing

To address the issue ③, eTran designs a new eBPF map `PKT_QUEUE` (i.e., `BPF_MAP_TYPE_PKT_QUEUE`), serving as the backbone data structure for various pacing mechanisms. A `PKT_QUEUE` contains multiple buckets, each with slots storing pointers to `xdp_frames`, which describe packet metadata (e.g., packet address). We leverage the eBPF `dynptr` feature (§2.2) to allow eBPF programs to safely access these `xdp_frames` by the pointers stored in `PKT_QUEUE` buckets (i.e., by functions `bpf_dynptr_from_xdp_frame` and `bpf_dynptr_slice_rd/rdwr`).

eTran's `PKT_QUEUE` supports enqueueing and dequeueing `xdp_frame` to/from a specified bucket. Depending on the pacing mechanism, the bucket index is determined based on the packet transmission timestamp (i.e., `tx_timestamp`) or explicitly specified; we will cover more detail at the end of this subsection. We further wrap a series of eBPF kfuncs (§2.2) to help eBPF programs manipulate `xdp_frame` pointers. For example, an existing kernel function `dev_xdp_enqueue` is wrapped into a kfunc to transmit packets through NIC queues.

In a quick summary, our `PKT_QUEUE` allows safely tracking packets deferred for transmission, and its associated kfuncs allow transmitting packets from eBPF programs. Now, the next question is: *how to asynchronously execute an eBPF program (that transmits packets)?*

BPF timers as triggers. To answer this question, eTran's approach is leveraging the mature "callback" infrastructure of BPF timers [67]. The original BPF timer works as follows: eBPF programs call `bpf_timer_init` to register an eBPF callback function, then use `bpf_timer_start` to start the timer with input parameters of the desired timeout value and some flags; when the timeout is reached, the eBPF callback

function will be executed asynchronously. Instead of using the timeout functionality of BPF timers, eTran leverages their callback registration and asynchronous execution abilities. Concretely, we extend BPF timers with two new modes by adding new flags to the `bpf_timer_start` function:

- In the first mode, this function will enqueue the callback function registered with this timer into a per-CPU backlog queue and schedule `NETTX_SOFTIRQ`. The softirq context at the same CPU would dequeue and execute callback functions based on the transport-specific pacing mechanism.
- In the second mode, the function enqueues the callback into the backlog queue of a dedicated kthread created at kernel boot, and wakes it up. The thread will dequeue and execute callbacks based on the specific pacing mechanism.

The first mode triggers per-CPU callback executionn (common for most protocols), while the second mode suits transport protocols that require global scheduling such as Homa. Both modes can be used together if callbacks properly synchronize shared states. In our implementation, TCP uses the first mode, while Homa uses both modes.

The remaining question is: *how to support various pacing mechanisms used in different transports?* We classify pacing mechanisms into several categories and discuss individually.

Rate-based pacing. This is the most widely-used pacing mechanism as it effectively prevents bursts and reduces congestion spikes [28, 44]. eTran leverages `PKT_QUEUE` to implement a scalable and CPU-efficient timing wheel, following the state-of-the-art design Carousel [62]. Specifically, each `PKT_QUEUE` bucket serves as a timing wheel slot, indicating when stored packets should be transmitted. For details on eTran's `tx_timestamp` mapping to buckets and timing wheel operation, see the Carousel paper [62]. Note that eTran also supports complementing rate-based pacing with window boundaries. Our TCP implementation adopts this pacing.

Credit-based pacing. This is widely used in receiver-driven transport protocols such as Homa. Take Homa as an example: when an RPC cannot send packets because of insufficient credits, Homa puts the RPC packets into a pacing engine and waits for the credits to replenish. To support credit-based pacing, eTran lets each `PKT_QUEUE` bucket store packets belonging to a Homa RPC; when the credit of a RPC is replenished, eTran will transmit packets in the corresponding bucket.

Out-of-order completion for AF_XDP. For the existing AF_XDP, packets are always completed (i.e., transmitted) in the order of being pushed into the TX ring, and its buffer recycling mechanism only supports in-order completion. However, with pacing introduced in `XDP_EGRESS`, the transmission of some packets will be delayed, causing out-of-order completion. To address this, we enhance the AF_XDP packet buffer management with minor code changes. See §5 for more detail.

3.3 eTran Data Path: User-Space IOs

eTran carefully made design choices inspired by user-space transports for IOs, which are critical for performance gain over

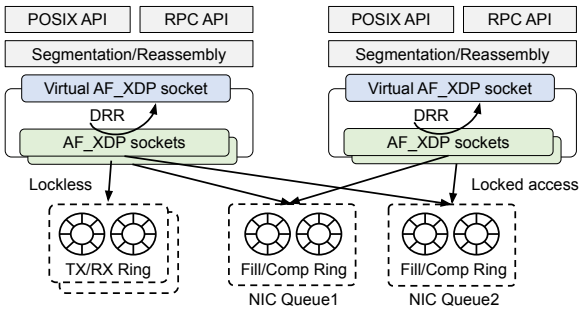


Figure 3: User-space packet IOs in eTran.

kernel-native transports. It offloads core transport operations to the kernel via eBPF and delivers data packets to the user space via AF_XDP sockets. A thin and untrusted user-space library reassembles raw data packets into transport-specific abstractions such as TCP flows or Homa RPCs and versa visa. However, we note that the untrusted library cannot directly access any transport state in the kernel, as it is within the non-privilege applications. Additionally, the library cannot access another application's UMEM (§3.1). This prevents untrusted libraries from corrupting kernel states or disrupting other applications, ensuring the transport stack remains protected.

Virtual AF_XDP socket. Figure 3 shows how our user-space library transforms AF_XDP packets into application-facing flows or RPCs. eTran utilizes multiple NIC queues, allowing different application threads to do packet IOs independently for high performance. Ideally, each application thread owns an AF_XDP socket accessing multiple NIC queues for higher parallelism and lower head-of-line blocking [13,37]; however, AF_XDP has an inherent constraint that one AF_XDP socket can only be bound to one NIC queue. To address this constraint, eTran proposes a *virtual AF_XDP socket* managing multiple real sockets (each bound to a NIC queue) via `epoll`.

Atop the virtual AF_XDP socket, eTran provides two types of APIs to fulfill the needs of different applications: 1) POSIX APIs with standard socket operations like `read` and `write`, and 2) event-driven RPC APIs with continuation callbacks like eRPC [38]. Applications using standard socket APIs can switch to eTran by LD_PRELOAD without any code change.

However, using virtual AF_XDP sockets also introduces new challenges on scheduling and thread synchronization.

Scheduling challenge. Packet arrival at a virtual AF_XDP socket indicates that packets have arrived at one or more real AF_XDP sockets. eTran must take care of how to pull packets from these real sockets in a load-balanced manner, especially to avoid starvation. To this end, eTran employs Deficit Round Robin (DRR) [66] scheduling to achieve efficient and fair packet processing among multiple sockets. When sending packets to real AF_XDP sockets, eTran operates similarly by enhancing the kernel `xsk_tx_peek_desc` function.

Synchronization challenge. All AF_XDP rings (i.e., TX/RX and Fill/Comp rings) are SPSC (single-producer, single-consumer) lockless rings designed for high-performance, run-to-completion packet processing. The TX/RX rings are per-

AF_XDP socket, therefore application and NAPI operations on them are still lockless. The Fill/Comp rings, however, are per-NIC queue. Multiple application threads can access them concurrently. A user-space spinlock is used to protect each Fill/Comp ring pair for correct concurrent access among application threads. No locking is needed for NAPI operations, as each NIC queue is handled by a single NAPI context.

Performance advantages of eTran. Compared to existing kernel transports such as Linux TCP and Homa kernel module, eTran has the following performance advantages: 1) efficient buffer management and syscall-free IO batching, 2) streamlined implementation without complicated kernel socket and (thus) file system, and 3) decoupled kernel transport processing[1], which reduces context switch [81] and improves instruction execution with better cache efficiency [41].

3.4 Multi-Tenancy and Traffic Management

eTran also offers software-based multi-tenancy and traffic management as a complement to hardware mechanisms (e.g., SR-IOV [17]). For different applications using eTran, the control path daemon isolates kernel resources (e.g., eBPF maps) by using different namespaces and assigns different NIC queues to different applications. This design prevents malicious applications from accessing others' IO buffers and enables operators to schedule problematic NAPIs on other CPUs to mitigate interruptions. Allocating at least one NIC queue per application is feasible, as modern NICs support hundreds to thousands of queues [68]. For applications not using eTran, XDP/AF_XDP naturally co-exists with other kernel built-in transports. We will demonstrate that eTran can operate seamlessly with kernel built-in transports in §6.3.

eTran uniquely excels in flexible traffic management as all packets traverse eBPF hooks. It natively supports traffic management such as traffic monitoring, access control, and rate limiting. eTran abstracts these tasks into *modular* eBPF programs, linked via `bpf_tail_call` that allows chaining multiple programs on a single hook. Operators can flexibly manage eBPF programs through the control path daemon.

4 Case Studies

This section takes two representative transports as examples to demonstrate how eTran enables implementing and customizing transport protocols in eBPF with high performance. The first transport is the widely-used TCP with DCTCP congestion control, which is a sender-driven and connection-orient protocol (§4.1). The second transport is Homa, which is a receiver-driven and connection-less protocol for datacenter RPCs (§4.2). These two transports represent a wide range of transport designs in literature [14, 26, 30, 44, 45, 49, 82]. We discuss more transport implementations in Appendix A. While eTran currently targets datacenter protocols pursuing performance, we believe it can be extended to non-datacenter scenarios such as wide-area networks. This would require more designs to handle more complex packet reordering and security mechanisms such as TLS [34] in eBPF.

[1]For Linux, the sender-side transport processing is coupled to the application context in kernel space when calling `write` or `send`.

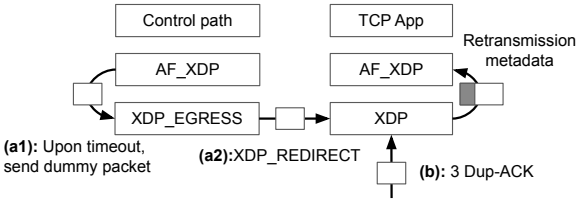


Figure 4: TCP retransmission in eTran. **(a)** is the retransmission triggered by the control path after a timeout, and **(b)** is the fast retransmission triggered by three duplicated ACKs.

4.1 TCP under eTran

TCP control path. As briefed in §3.1, the control path daemon manages TCP connections, congestion control (CC), and severe loss recovery when timeout happens. The daemon uses its own AF_XDP sockets to communicate with remote peers for the three-way handshake to establish TCP connections. It then installs TCP connection states and CC states into eBPF maps. Separating CC states into the control path is inspired by the CCP architecture [52] with two motivations: 1) eBPF does not have mature support for floating point computation that commonly appears in TCP CC algorithms, and 2) CCP has shown that separating CC from the datapath still gives similar CC behaviors and accuracy as in datapath. We defer the description of loss recovery to the end of this subsection, as it interacts closely with the data path.

eTran maintains the connection states in an eBPF hashmap while CC states are in an eBPF array. Storing CC states in an array allows eTran to `mmap` it into the daemon's address space through the `BPF_F_MMAPABLE` feature; this avoids the syscall overhead when the daemon updates CC states. To help eBPF programs locate the CC state, each connection state contains the index of the corresponding CC state in the array.

TCP data path. The eBPF program at the XDP hook receives both TCP control packets and data packets: it redirects control packets such as `SYN` and `SYN-ACK` to the control path AF_XDP; for data packets, it uses four-tuple as the key to look up TCP connection state in the aforementioned eBPF hashmap. The eBPF program then verifies the `ack` and `seq` values of data packets and updates TCP windows if valid. For common cases where `seq` is in order, packets are delivered to the virtual AF_XDP sockets in the user-space transport library (see §3.3). For out-of-order `seq`, eTran implements an out-of-order receiving mechanism similar to TAS [41].

Figure 4 shows eTran's workflow for loss recovery, including fast retransmission and timeout-triggered retransmission. To handle fast retransmission, once the eBPF program at the XDP hook detects three duplicate ACKs, it will rollback the transport state, piggyback the retransmission information (i.e., the rollback pointer) in the ACK packet, and deliver the ACK packet to user-space transport library. The library

will retransmit packets accordingly. The piggybacked retransmission information is stored in the headroom of the packet frame, enabling efficient coordination between the eBPF program and the user-space transport library. To handle timeout-triggered retransmission, once the control path daemon detects such timeout, it will send a dummy packet going through the `XDP_EGRESS` hook to trigger the fast retransmission logic of the eBPF program and then get dropped. This avoids locking operations between the data path and the control path.

The eBPF program at the `XDP_EGRESS` hook processes every packet transmitted by the AF_XDP sockets. It fills packets with the right TCP header and IP header based on the connection state maintained in eBPF maps and the right Ethernet header by looking up the kernel FIB table with `bpf_fib_lookup`. To reduce the cost of looking up the FIB table for every packet, eTran uses an eBPF array to cache results. When the flow control window and the CC window both permit, the `XDP_EGRESS` eBPF program will transmit the packet directly with the action of `XDP_TX`; otherwise, it will enqueue the packet into our `PKT_QUEUE` with the action of `XDP_REDIRECT`. Later, when both windows permit, the queued packets will be transmitted asynchronously (see §3.2.2). As briefed in §3.2.1, this eBPF program also prevents misbehaved applications from modifying connection states that do not belong to them. For example, it will check if the `umem_id` in the packet context matches the one maintained in the connection state and drop the packets if not matched.

4.2 Homa under eTran

Homa control path. Similar to TCP, Homa under eTran lets the control path daemon manage its socket creation, binding, and closing. Different from TCP, Homa runs CC in the data path via eBPF (rather than in the control path) with CC states maintained in eBPF hashmaps. There are two reasons for such a difference: 1) Homa CC or its receiver-driven credit generation is in the critical path of data packet transmission (in contract to TCP sender-driven CC that regulates rate), thus having huge performance penalty if running on a separate control path; and 2) Homa CC is relatively simple in terms of compute operations (e.g., no floating points), friendly to eBPF offloads. Since Homa is connection-less via short-lived RPCs, eTran also maintains its RPC states in the data path for efficient creation/destruction (with eBPF hashmaps).

The Homa control path detects RPC timeout and handles it by sending `RESEND` packets for retransmission. The daemon's timer thread periodically wakes up and scans RPC states in the kernel eBPF hashmap for detecting timeouts; it uses `bpf_map_lookup_batch` to reduce syscall overhead. Once the peer receives the `RESEND` in the XDP hook, XDP will piggyback the lost range and RPC buffer address (maintained in the kernel RPC state) on the packet and redirect it to the user-space library through AF_XDP for retransmission.

Other control-path events include maintaining infrequently changing priority cutoff values [55], freeing states in remote RPC servers when RPC clients do not explicitly confirm RPC completions, and aborting RPCs caused by machine crushes or long timeout by forcibly removing corresponding states.

Homa data path. Homa uses receiver-driven credit scheduling to achieve SRPT (Shortest Remaining Processing Time); eTran implements the credit scheduling at the XDP hook, while also enforcing sender-side SRPT at the `XDP_EGRESS` hook. When the XDP eBPF program receives the first *unscheduled packet* [50] of an RPC message, it creates a new CC state object, including the RPC ID and remaining bytes through `bpf_obj_new`, and enqueues it into a `bpf_rbtree` map, i.e., *credit list*. For subsequent packet arrivals, the eBPF program traverses the credit list, dequeues the corresponding state object, updates the state object with new remaining bytes, and re-enqueues it into the credit list.

The eBPF program at the `XDP_GEN` hook is responsible for selecting RPCs and sending credit packets. Conceptually, it has the following four steps: 1) selecting candidate RPCs from the credit list; 2) looking up the RPC state hashmap to find corresponding RPC states, then allocating credits; 3) removing finished RPCs from the credit list; 4) sending credits for the selected RPCs. However, we face two challenges due to eBPF constraints: lack of searching functionality in `bpf_rbtree` to implement RPC selections; instruction limit posed by the eBPF verifier disallows offloading complex logic.

eTran addresses these challenges by innovatively utilizing existing eBPF features. For the first challenge, we wrap a new eBPF kfunc called `bpf_rbtree_lower_bound` that returns the first node whose value is not less than the provided one; we then leverage this kfunc to select candidate RPCs ranked by their priorities while guaranteeing not selecting more than one RPC for each peer (required by Homa [55]). A common way to implement such RPC selections is through a 2-level rbtree as Homa does, but requires holding multiple locks simultaneously, which eBPF prohibits. Instead, we use `bpf_rbtree_lower_bound` to emulate such 2-level rbtree with a single rbtree; see Appendix B for more detail. For the second challenge, we use `bpf_tail_call` to break complex eBPF logic into multiple programs that go through the verifier separately. To pass states across different tail-called eBPF programs, we exploit per-CPU variables in the `bss` section.

On the sender side, the `XDP_EGRESS` eBPF program queues packets into the pacing engine (see §3.2.2) when no credit is available for this RPC. Upon credit replenishment, the callback enforces SRPT across RPCs using a `bpf_rbtree` of queued RPCs, sorted by the remaining bytes to send.

Due to space limitation, we elaborate on how eTran enables a flexible development model covering transport developers, application developers, and more in Appendix C.

5 eTran Implementation

Our eTran prototype contains 24K lines of C/C++, spreading across the Linux kernel, control path daemon, eBPF programs, and user-space transport library, as shown in Table 1. The

Category	Kernel	Control path daemon	eBPF programs: TCP/Homa	Transport libs: TCP/Homa
LoC	2597	8224	2173/5349	3661/2024

Table 1: eTran codebase breakdown with 24028 LoC in total.

current prototype runs on Linux kernel v6.6.0 and targets Mellanox mlx5 driver. Our Linux kernel change includes the two new hooks, the new eBPF map with associated kfuncs (for packet manipulation), `bpf_rbtree`'s new API as a kfunc, and AF_XDP's out-of-order completion with mlx5 driver support. We only make ~20 LoC change for the mlx5 driver, so that the driver can maintain the addresses of packet frames that get queued in the pacing engine, and only place them in the completion ring when they are eventually transmitted.

eBPF verifier. eTran does not change the verifier but just registers new eBPF hooks and kfuncs with the verifier for safety and security. In general, there are three types of checks done by the eBPF verifier: (a) eBPF program behaviors (e.g., bounded loops, NULL pointers, deadlock, etc.), (b) how eBPF programs access input contexts, and (c) how eBPF programs use eBPF helpers and kfuncs. Naturally, the eBPF verifier checks (a) for all eTran eBPF programs. It also checks (b) for `XDP_GEN` and `XDP_EGRESS` similarly to the existing XDP.

For (c), the principle is that eBPF programs must call these helpers and kfuncs safely and securely. For safety, we ensure the used kfuncs (either wrapped from the existing kernel functions or written by us) are non-preemptible and have finite execution steps. For example, our `XDP_EGRESS` hook uses the kfunc wrapped from the kernel `dev_xdp_enqueue()` function, which meets the safety requirement as it was designed to run in the softirq context. Our added kfunc for `bpf_rbtree` follows the implementation practice of existing eBPF helpers with extensive testing; nevertheless, we plan to verify it formally in the future. For security, we expose only necessary eBPF helpers to our new hooks and carefully reason through its security implications. The eBPF helpers exposed to the new hooks are only a `subset` of the ones exposed to XDP by kernel developers. For example, we exclude all socket-related helpers (e.g., `bpf_sk_lookup_tcp`) as the new hooks don't access these sockets; we exclude `bpf_redirect_map` for `XDP_GEN` to prevent redirecting packets arbitrarily.

6 Evaluation

This section aims to answer the following questions:

1. What are the latency and throughput of TCP and Homa under eTran compared to existing kernel ones and kernel-bypass ones (§6.1 and §6.2)?
2. How well does eTran support multi-tenancy and traffic management (through our new eBPF map) (§6.3)?
3. What is the performance of our new eBPF hooks (§6.4.1) and retransmission design (§6.4.3)?

Experiment setup. We use 10 `xl170` physical machines from CloudLab [19], each equipped with two 10-core Intel E5-2640v4 CPUs (2.4GHz), 64GB memory, and a Mellanox ConnectX–4 25 Gbps NIC. All machines are connected via a Mellanox 2410 switch under the same rack. For eTran (Homa/TCP) and Linux (Homa/TCP), all machines run Linux kernel v6.6.0. For TAS, we are unable to run it on Linux kernel v6.6.0 due to Mellanox driver issues, but we manage to run it on Linux kernel v5.15.0. Since TAS is built atop DPDK, we expect no performance differences between the two kernel versions. We only enable TCP Segmentation Offload (TSO) for Linux (Homa/TCP) as AF_XDP does not currently support it (see §7). There is no zero-copy optimization for eTran for a fair comparison (i.e., keep API semantics the same). Following prior work [80, 81], we disabled the NIC interrupt coalescing feature. All experiments use the default maximum packet size (MTU) of 1500 bytes. For DCTCP, we set the ECN marking threshold to 70KB, as done in prior work [55].

Comparison baselines. We compare eTran (Homa) to Linux/Homa (commit `8321cde`), a kernel module implementation of Homa that is highly optimized by the Homa authors. We refer to it as Linux (Homa) throughout this paper. We compare eTran (TCP) to TAS (commit `d3926ba`), a microkernel-style user-space TCP stack built on DPDK, and Linux (TCP) with DCTCP congestion control. By default, for eTran and TAS, we separate CPU cores for applications and interrupt processing/busy polling, and provision a dedicated core for control path/slow path. Linux (Homa) and Linux (TCP) use all provisioned cores to handle network load and applications.

6.1 Comparision with Linux (Homa)

Microbenchmarks. We first compare the basic latency and throughput between eTran (Homa) and Linux (Homa). We use a similar configuration described in the Homa paper [55]. The median latency of short messages and the throughput of large messages are measured by using a single client thread to send back-to-back requests (32B/1MB) to a single-threaded server, which responds with a 32-byte response. Next, we measure the throughput of a multi-threaded server receiving concurrent RPCs (500KB) from 7 clients, as well as the throughput of a multi-threaded client sending concurrent RPCs to 7 servers. Finally, we measure the RPC rate for small messages (32B), maintaining the same client-to-server ratio.

Table 2 summarizes the result. eTran (Homa) achieves lower median latency than Linux (Homa), i.e., 11.8μs vs. 15.6μs. eTran (Homa) outperforms Linux (Homa) in large message throughput, even with Linux (Homa) using TSO batching. In terms of RPC rate, eTran (Homa) achieves approximately 1.7-1.8$\times$ the message throughput of Linux (Homa). This gap is because eTran (Homa) uses AF_XDP for efficient packet IO while Linux (Homa) has a larger code path and uses costly structures like `sk_buff`.

Cluster benchmark. All the following experiments utilize the same cluster benchmark application from Linux (Homa) [55], conducted with 10 machines. In this experiment, each node serves as both a multi-thread client and a multi-thread server simultaneously. Clients randomly select servers to issue a batch of RPCs. Given that the data of modern RPCs primarily

	eTran (Homa)	Linux (Homa)
32B median latency (μs)	11.8	15.6
1MB throughput (Gbps)	17.7	14.5
Server throughput (Gbps)	23.0	23.1
Client throughput (Gbps)	22.7	22.9
Client RPC rate (Mops)	2.9	1.7
Server RPC rate (Mops)	3.3	1.8

Table 2: Comparison of basic latency and throughput between eTran (Homa) and Linux (Homa).

flows in one direction [64], we vary the request size up to 1MB while keeping the response size fixed as a small message. We use workloads W2-W5 in Homa paper [55] where W2 and W3 are dominated by short messages and W4 and W5 are dominated by large messages. Slowdown is the client-observed RTT divided by the ideal RTT for the same-length RPCs using eTran (Homa). For fairness, eTran (Homa) is configured with the same parameters as Linux (Homa).

Figure 5 shows the latency slowdown for workloads W2-W5. eTran (Homa) outperforms Linux (Homa) across all workloads. For the short message dominated workloads W2 and W3, eTran (Homa) achieves 3.9-7.5× lower P99 tail latency and 1.4-3.6× lower P50 latency compared to Linux (Homa). This is because: for W2 and W3 where the bottleneck is software overhead, eTran (Homa) leverages AF_XDP to achieve more efficient packet processing. For W4 and W5, which mainly consist of large messages, eTran (Homa) also slightly outperforms Linux (Homa). eTran (Homa) occasionally experiences higher tail latency than Linux (Homa) for large messages. This is because less optimized thread scheduling in eTran (Homa) causes interference between the pacing thread and applications; Linux (Homa) also faced this issue but mitigated it by carefully scheduling both threads to ensure pacing neither falls behind nor dominates CPUs. We plan to incorporate such transport-specific scheduling policies in the future, possibly through eBPF as well [63].

Figure 6 shows the RTT distributions for the shortest messages (10%) in W4 and W5, where eTran (Homa) achieves lower latencies across almost all percentiles. For example, eTran (Homa) achieves 4.1× lower P50 latency than Linux (Homa) in W4 and 3.9× lower in W5. The P99 latency is reduced by 4.3× in W4 and 2.9× in W5.

6.2 Comparision with Linux (TCP) and TAS

We built two POSIX API-based applications: a simple echo server and a key-value store derived from the TAS codebase (modeled after Memcached). The server handles requests using `epoll()`, and the client generates requests in a closed loop. All systems use the same binary but link to different POSIX implementations (e.g., Linux, TAS, and eTran).

Message throughput. Figure 7a shows the throughput of transferring a single back-to-back large message. eTran (TCP) performs comparably to Linux (TCP) but slightly lags behind TAS; for messages under 64KB, eTran (TCP) surpasses Linux (TCP). Figure 7b shows the throughput for small messages

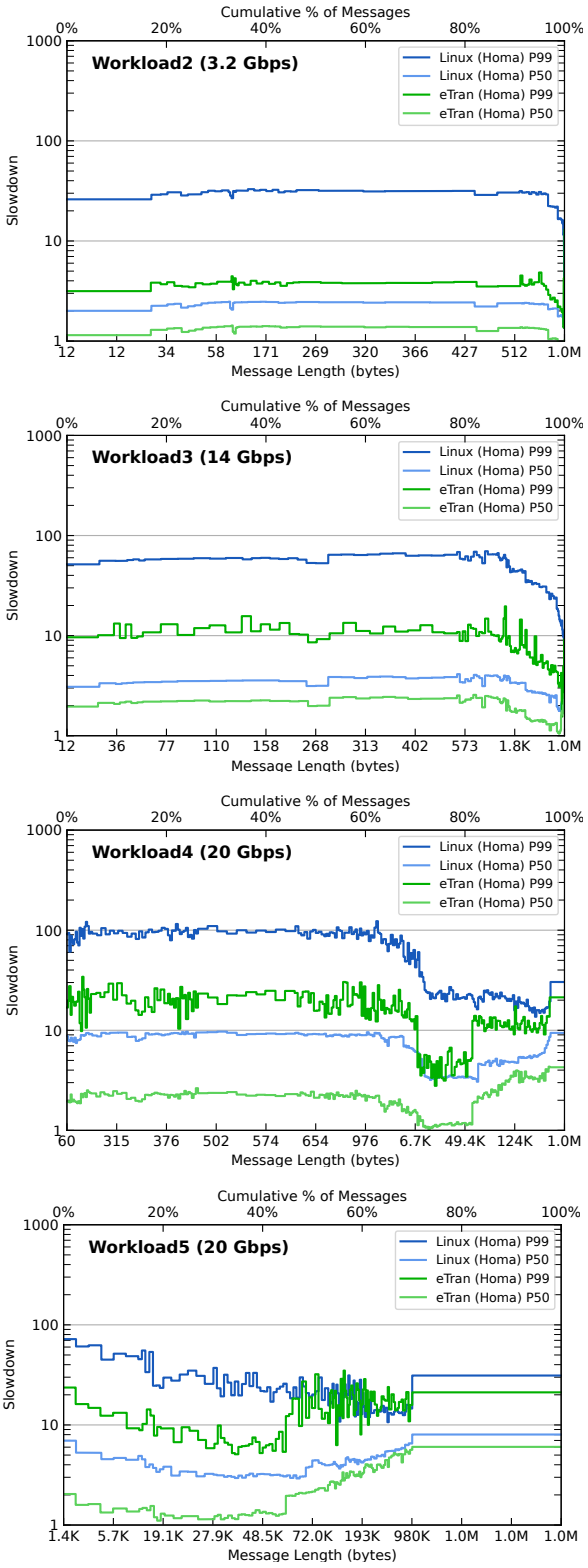


Figure 5: Median and 99th-percentile latency slowdown vs. message size for workloads W2–W5. The x-axis is scaled linearly to represent the number of RPCs, reflecting each workload's CDF of message lengths. W2 and W3 are dominated by short messages, while W4 and W5 feature larger messages.

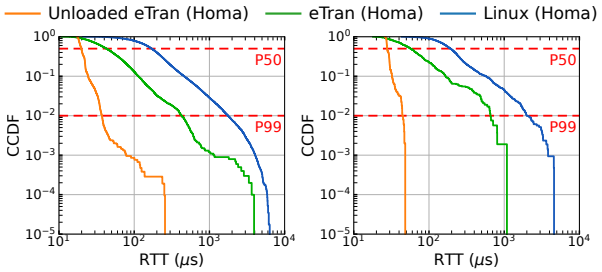
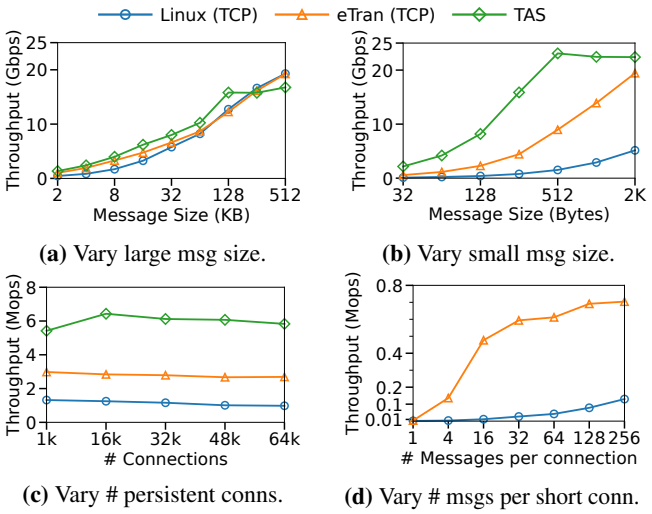


Figure 6: Complementary CDF of round-trip latency for the shortest 10% of RPCs in W4 (left) and W5 (right).



(a) Vary large msg size. **(b)** Vary small msg size.

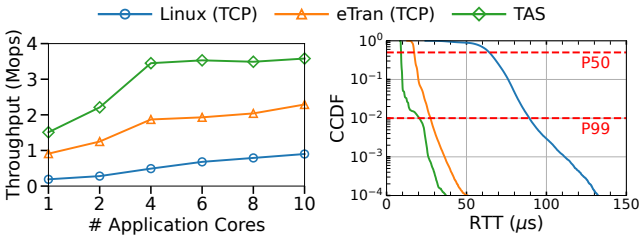
(c) Vary # persistent conns. **(d)** Vary # msgs per short conn.

Figure 7: Comparison of message throughput for the TCP echo. Figure (a) has a single outstanding message, while Figure (b) has 64. TAS fails to achieve stable throughput for Figure (d), thus omitted.

with 64 outstanding messages per connection. This is measured using a single-threaded server handling 100 connections, partitioned across five clients each with five threads. For example, with 1KB messages, eTran (TCP) and TAS achieve 4.8× and 7.7× higher throughput than Linux (TCP), respectively. The gap between eTran (TCP) and TAS stems from TAS's use of dedicated cores for DPDK-based busy polling, while eTran (TCP) is interrupt-driven. Nevertheless, eTran (TCP) consistently outperforms Linux (TCP) across all message sizes and achieves 87% of TAS throughput for 2KB messages.

Connection scalability. We use as many clients as possible to generate the load with persistent connections. Clients send a single 64B request per connection and wait for a response in a closed loop. Figure 7c shows the result. The throughput of eTran (TCP) falls between Linux (TCP) and TAS: e.g., with 1K connections, eTran (TCP) achieves 2.26× the throughput of Linux (TCP), while TAS achieves 4.1× the throughput of Linux (TCP). Linux suffers high syscall overhead with limited TSO/GRO benefits with many connections. eTran (TCP) batches processing but is still less efficient than TAS due to the performance gap between AF_XDP and DPDK [39].

Short-lived connections. In this experiment, the client uses four threads to establish a total of 1K short-lived concurrent flows with a single-threaded server. Upon receiving the



(a) KV tpt vs. #app cores. **(b)** CCDF of KV RTT.

Figure 8: Performance comparison of the TCP key-value store.

server's response for each connection, the client immediately closes the connection and establishes a new one. The result for TAS is omitted as it fails to achieve stable throughput due to its implementation bugs. As shown in Figure 7d, with one message per connection, the throughput of eTran (TCP) and Linux (TCP) is similar. With four or more messages per connection, eTran (TCP) significantly outperforms Linux (TCP). For example, eTran (TCP) achieves 42.7× and 5.4× the throughput of Linux (TCP) with 16 and 256 messages per connection, respectively. This is because the eTran (TCP) control path uses AF_XDP to perform efficient packet IOs for connection handshake and teardown, while the data path utilizes efficient LRPC to communicate with the control path.

Key-Value Store. We use a workload that has 100K key-value pairs (32B keys, 64B values) following a Zipf (s=0.9) distribution, with a 9:1 GET to SET ratio. Each of the five client machines establishes 6K persistent connections, leaving 32 outstanding requests per connection. Figure 8a shows that eTran (TCP) achieves 2.4-4.8× the throughput of Linux (TCP), with lower performance than TAS (which achieves 3.9-7.9× of Linux). Figure 8b provides more detailed insights into the latency when the server is under-loaded: eTran (TCP) has slightly higher latency than TAS but is significantly lower than Linux (TCP). More specifically, eTran (TCP) achieves 3.7× (17.2 µs vs. 64.2 µs) lower P50 latency and 3.2× (27.5 µs vs. 89.3 µs) lower P99 latency compared to Linux (TCP).

6.3 More Features

6.3.1 Multi-Tenancy and Coexistence

eTran (TCP/Homa) supports multi-tenancy and can coexist with Linux (TCP/Homa), thanks to XDP/AF_XDP not taking over the entire NIC. Figure 9 shows the throughput timeline when using different protocols. Launching our control path slightly reduces Linux throughput due to the extra XDP processing. This overhead can be removed by loading XDP on isolated queues from Linux (TCP/Homa) [71] and using hardware mechanisms (e.g., Flow Director [16]) for isolation.

6.3.2 Traffic Management

As an example, we show that eTran enforces accurate rate limiting with low overhead using `PKT_QUEUE` and pacing engine (§3.2.2). eTran tail-calls a simple eBPF program after `XDP_EGRESS` to retrieve the target rate from a policy table and

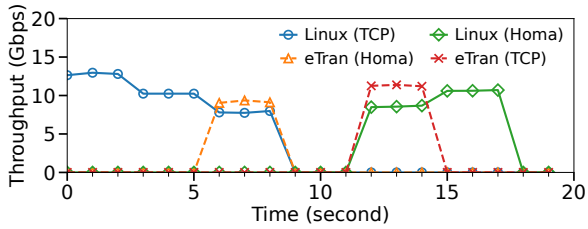
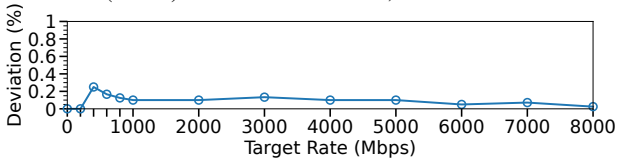
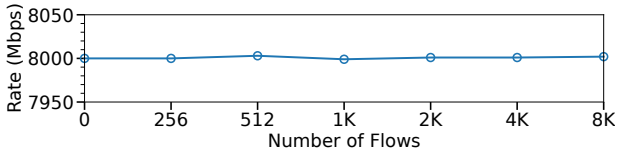


Figure 9: At 0 sec, APP1 starts TCP flows. At 3 sec, eTran launches the control path and loads eBPF. At 5 sec, APP2 sends RPCs with eTran (Homa). At 9 sec, both apps exit, and the control path switches protocols. At 12 sec, APP3 runs eTran (TCP), and APP4 sends RPCs with Linux (Homa). APP3 exits at 15 sec, and APP4 at 18 sec.



(a) The deviation from the target rate of eTran for a single flow.



(b) The actual rate achieved by eTran with a target rate of 8Gbps.

Figure 10: Rate conformance of eTran.

calculate the timestamp; it then uses this timestamp in the pacing engine to regulate the traffic rate. Figures 10a and 10b show rate conformance for single and multiple flows, with deviations under 0.4% as the rate and flow count increase.

6.4 Design Drill-Down

6.4.1 eBPF Hook Performance

We show that our new hooks offer powerful functionalities with minimal overhead. We employ three benchmarks from `bpf-examples` [9]: `tx-only` for small packet transmission, `l2fwd` for layer 2 forwarding, and `rx-drop` for packet discarding. 64B packets are used to stress test the performance.

XDP_EGRESS overhead. Table 3 shows that an empty hook reduces throughput by 6.6%. Out-of-order completion further lowers it to 86.1%. The primary overhead in `XDP_EGRESS` comes from constructing new packet buffers, which is mitigated by pre-allocations (§3.2.1). The overhead of out-of-order completion comes from higher cache miss rate incurred by increased memory footprint. As a reference, the table also notes the cost of map lookup operations in `XDP_EGRESS`.

XDP_GEN performance. We run `rx-drop`+XDP_GEN on two cores: one core generates ACK/credit packets using XDP_GEN, and another uses AF_XDP to drop received packets. We compare it to `l2fwd` on two cores: one core in the kernel redirects packets to AF_XDP, and another uses AF_XDP to forward packets back as ACK/credit packets. Table 4 shows that our overall throughput is slightly lower than `l2fwd` but with higher per-core throughput. This is because we avoid cross-core communication for ACK/credit packet transmission, as it is done inside the kernel. We conclude that `XDP_GEN` is efficient enough in transmitting ACK/credit packets.

Action	Tpt (Mpps)	Tpt loss (%)
AF_XDP `tx-only`	11.55	-
+ Empty XDP_EGRESS	10.79	6.6
+ OOO comp	9.95	13.9
+ ARRAY lookup	9.71	15.9
+ HASHMAP lookup	9.10	21.2

Table 3: Performance of XDP_EGRESS (single core).

	Tpt (Mpps)	# cores	Per-core tpt (Mpps)
`l2fwd`	6.73	1.74	3.87
`rx-drop` + XDP_GEN	6.03	1.35	4.47

Table 4: `l2fwd` vs. `rx-drop` + XDP_GEN on receiving data packets and responding with ACK/credit packets.

6.4.2 CPU Overhead Breakdown

Table 5 presents the CPU overhead across the networking stack for a multi-threaded server. We stress-test transport processing with a single NAPI, isolating it from applications explicitly for all systems. We made great efforts to tune the Linux stacks to be even better than the one reported in [55] under the same hardware. Note that this configuration is not general for all applications. Per-function cycles are obtained using Perf [29] and categorized with tools extended from [12]. Homa incurs higher overheads than TCP in both implementations due to frequent allocation/destruction for small RPCs. eTran significantly reduces primary overhead (*Socket/RPC*, *TCP/Homa+IP*) through streamlined implementation, with optimizations to *Sk_buff*, *Memory Mgmt* via lightweight xdpbuff and pre-allocated UMEM. *Scheduling* is optimized through batched IO and decoupled kernel transport processing. For Linux, per-request context switching incurs higher instruction counts, pipeline stalls, memory footprint, and poor cache locality [41]. *Other* overheads (e.g., security checks, kernel/user mode switching) are reduced due to file system bypassing and AF_XDP's syscall-free asynchronous IO.

6.4.3 Impact of Retransmission Design

We show that retransmission handling in the control path matches Linux (Homa) and Linux (TCP) performance. By launching 100 flows between two machines and measuring throughput under varying loss rates, Figure 11 shows the throughput penalty of different protocols. eTran (TCP) incurs a larger throughput penalty than TCP due to limited out-of-order optimization. Homa's retransmission is simple as packet loss is rare; both eTran (Homa) and Linux (Homa) show similar throughput decline. Due to space constraints, we show additional experiments on control path CPU usage, multi-queue impact, and interrupt interference in Appendix D.

7 Discussion and Future Work

Emerging kernel techniques. Many emerging kernel techniques inspire eTran design or might help improve eTran per-

Component	eTran (T)	Linux (T)	eTran (H)	Linux (H)
Application	0.48	0.53	0.95	1.04
Socket/RPC	0.63	**3.5**	0.98	**3.38**
Data copy	0.19	0.57	0.32	1.30
Sk_buff	0.15	0.47	0.08	0.39
TCP/Homa+IP	1.06)	**2.12**	1.47	**3.36**
Lock/Unlock	0.18	0.45	0.24	2.68
NIC Driver	1.17	1.54	0.83	1.81
Memory Mgmt	0.05	0.32	0.06	1.04
Scheduling	0.25	1.19	0.18	1.02
Other	0.21	1.82	0.38	1.41
Total (kcycles)	**4.37**	**12.51**	**5.48**	**17.43**

Table 5: Breakdown of CPU Cycles per Request. **T**: TCP; **H**: Homa.

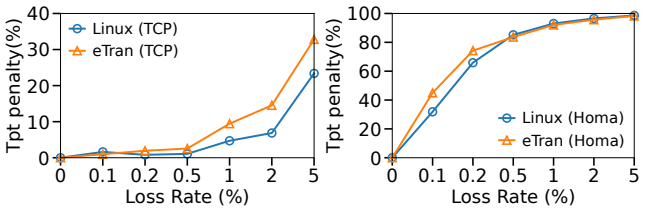
formance. For example, the Linux kernel community made efforts to introduce queueing for XDP when redirecting among different network interfaces [33]; instead, our pacing engine is to provide a general queueing mechanism for packet transmission. We also note that the existing Linux qdisc (queueing discipline) cannot sustain the high packet rate as our pacing engine [62]. From Linux kernel v6.6, some NIC drivers support AF_XDP multi-buffer [22], enabling features like TSO and scatter/gather. Unfortunately, the mlx5 driver we are using does not support them yet; we expect it could help improve eTran's performance for large messages/long flows.

Improving eBPF programming. eTran suffers from limited synchronization support in eBPF programming. For example, eBPF prohibits holding two locks simultaneously, posing significant challenges for implementing Homa's credit list (§4.2). Another example is that eBPF lacks blocking synchronization primitives like Mutex, necessitating careful spinlock use. We understand this limitation is rooted in the fact that the eBPF execution is non-sleepable and non-preemptable to simplify safety verification, but any relaxation of it would hugely simplify eBPF programming for complex applications.

eTran limitations. eTran relies on the eBPF subsystem to customize transports, suffering from similar limitations as eBPF. For example, eBPF does not support floating point arithmetic, motivating eTran to adopt the CCP architecture to separate CC into the control path daemon for sender-driven transports. Currently, eTran depends on the kernel for CPU scheduling, which can lead to tail latency issues, as noted in prior works [24, 54]. In addition, since AF_XDP bypasses kernel layers, eTran lacks the integration of other kernel subsystems such as file system. We leave them for future work.

8 Related Work

High-performance network transports. There has been a line of work on high-performance transports, e.g., mTCP [36], IX [4], Arrakis [57], TAS [41], eRPC [38], Snap [47], Demikernel [78] and Junction [23]. Most rely on kernel-bypass, busy polling to achieve high performance and co-locate with applications in the same process context (*library OS*). In contrast, eTran transports operate within the kernel interrupt-driven NAPI context, with protection ensured by eBPF. This trades performance for CPU efficiency and prevents applications



(a) Comparison of TCP. **(b)** Comparison of Homa.

Figure 11: Throughput penalty at varying packet loss rates.

from interfering with transport processing and states. Both IX and Arrakis provide transport protection for *library OS*. However, IX repurposes virtualization hardware non-standardly, while Arrakis relies on NIC virtualization, which is more limited and inflexible than software-based approaches. TAS and Snap separate the kernel-bypass transport on isolated cores from applications (*Microkernel*). Although retaining flexibility, it must reimplement many functionalities from scratch. Conversely, eTran still uses some kernel networking infrastructures, and new transports can co-exist with existing kernel ones. eTran also differs from *kernel* transports: eTran transports are split to the control path (user-space) and the data path (kernel eBPF); the untrusted library uses AF_XDP socket for efficient IO and metadata exchange (rather than relying on syscalls), while being restricted from directly accessing kernel transport states maintained in eBPF. This design enables safely customizing high-performance kernel transports.

eBPF verification. eTran relies on verification techniques to guarantee the safety of running application-supplied eBPF programs inside the kernel. Therefore, a bunch of eBPF verification work in this space such as [8, 40, 46, 69, 73, 76] would benefit eTran and enhance our kernel safety and security.

Kernel offloads. Offloading application functions to the kernel via eBPF has recently emerged to avoid kernel overheads and improve performance [11, 27, 37, 56, 59, 79–81]. Different from offloading applications via eBPF, eTran targets making kernel transport extensible by extending eBPF subsystems.

Extensible kernels. eTran adopts a similar approach that is reminiscent of the extensible kernel research in the 1990s such as SPIN [7], Exokernel [21], and VINO [65]. The key difference is that eTran relies on existing kernel eBPF subsystem for safety and targets performance-hungry network transports in modern datacenters. There is a line of recent work on making kernel TCP more extensive via eBPF [10, 31, 35, 70, 77]. However, they only focus on the TCP transport and allow customizing a small portion of TCP behaviors such as initial congestion window, CC schemes, and path management.

9 Conclusion

eTran is an extensible kernel transport system, achieving agile customization, kernel safety, strong protection, and high performance simultaneously. It achieves these goals by extending the kernel-safe eBPF, hiding transport states inside the kernel, and absorbing techniques from user-space transports. eTran code is available at https://github.com/eTran-NSDI25/eTran.

Acknowledgments

We thank our shepherd Boris Pismenny and the anonymous reviewers for their helpful comments. We thank Cloudlab [19] for the development and evaluation infrastructure. Zhongjie Chen, Yifan Liu, and Fengyuan Ren are supported in part by the National Key Research and Development Program of China (No. 2022YFB2901404), and by the National Natural Science Foundation of China (NSFC) under Grant No. 62132007 and No. 62221003. ChonLam Lao, Minlan Yu, and Yang Zhou are supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Yang Zhou is also supported by a Google PhD Fellowship and the UC Berkeley Sky Computing Lab.

References

[1] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the Performance Limits of Datacenter Networks. In *Proceedings of USENIX NSDI*, pages 51–70, 2022.

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of ACM SIGCOMM*, page 63–74, 2010.

[3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *Proceedings of USENIX NSDI*, pages 93–109, 2020.

[4] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of USENIX OSDI*, pages 49–65, 2014.

[5] Steven M Bellovin. Security Problems in the TCP/IP Protocol Suite. *ACM SIGCOMM CCR*, 19(2):32–48, 1989.

[6] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight Remote Procedure Call. *ACM SIGOPS Operating Systems Review*, 23(5):102–113, 1989.

[7] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of ACM SOSP*, pages 267–283, 1995.

[8] Sanjit Bhat and Hovav Shacham. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis, 2022.

[9] The bpf examples authors. bpf-examples. https://github.com/xdp-project/bpf-examples.

[10] Lawrence Brakmo. TCP-BPF: Programmatically Tuning TCP Behavior through BPF. *Proc. NetDev*, 2:1–5, 2017.

[11] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A Database Proxy That Bounces with User-Bypass. *Proceedings of the VLDB Endowment*, 16(11):3335–3348, 2023.

[12] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of ACM SIGCOMM*, pages 65–77, 2021.

[13] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *Proceedings of ACM SIGCOMM*, pages 767–779, 2022.

[14] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of ACM SIGCOMM*, page 239–252, 2017.

[15] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale RPCs with breakwater. In *Proceedings of USENIX OSDI*, pages 299–314, 2020.

[16] Intel Corporation. Introduction to Intel Ethernet Flow Director and Memcached Performance. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf.

[17] Intel Corporation. Single Root I/O Virtualization (SR-IOV) for Intel NICs. https://www.intel.com/content/www/us/en/support/articles/000005722/ethernet-products.html.

[18] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, et al. A Large-Scale Deployment of DCTCP. In *Proceedings of USENIX NSDI*, pages 239–252, 2024.

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of cloudlab. In *Proceedings of USENIX ATC*, pages 1–14, 2019.

[20] Kumar Kartikeya Dwivedi. Allocated objects, BPF linked lists. https://lwn.net/Articles/915404/.

[21] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[22] Maciej Fijalkowski. xsk: multi-buffer support. https://lwn.net/Articles/937525/.

[23] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making Kernel Bypass Practical for the Cloud with Junction. In *Proceedings of USENIX NSDI*, pages 55–73, 2024.

[24] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of USENIX OSDI*, pages 281–297, 2020.

[25] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of ACM SIGCOMM 2024 Conference*, page 57–70, 2024.

[26] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed Near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the ACM CoNEXT*, 2015.

[27] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of USENIX NSDI*, pages 487–501, 2021.

[28] Google. Falcon transport protocol. https://github.com/opencomputeproject/OCP-NET-Falcon.

[29] Brendan Gregg. Perf Examples. https://www.brendangregg.com/perf.html.

[30] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of ACM SIGCOMM*, page 29–42, 2017.

[31] Jörn-Thorben Hinz, Vamsi Addanki, Csaba Györgyi, Theo Jepsen, and Stefan Schmid. TCP's Third Eye: Leveraging eBPF for Telemetry-Powered Congestion Control. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, pages 1–7, 2023.

[32] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A Building Block for Proactive Transport in Datacenters. In *Proceedings of ACM SIGCOMM*, page 422–434, 2020.

[33] Toke Høiland-Jørgensen. Adding packet queueing to XDP. https://lpc.events/event/16/contributions/1351/attachments/1049/2037/xdp-queueing.pdf.

[34] IETF TLS Working Group. Transport layer security. https://datatracker.ietf.org/wg/tls/about/.

[35] Mathieu Jadin, Quentin De Coninck, Louis Navarre, Michael Schapira, and Olivier Bonaventure. Leveraging eBPF to Make TCP Path-Aware. *IEEE Transactions on Network and Service Management*, 19(3):2827–2838, 2022.

[36] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of USENIX NSDI*, pages 489–502, 2014.

[37] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-Defined Scheduling Across the Stack. In *Proceedings of ACM SOSP*, pages 605–620, 2021.

[38] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of USENIX NSDI*, pages 1–16, 2019.

[39] Magnus Karlsson and Björn Töpel. The path to dpdk speeds for af xdp. In *Linux Plumbers Conference*, volume 37, page 38, 2018.

[40] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, Flexible, and Practical Kernel Extensions. In *Proceedings of ACM SOSP*, 2024.

[41] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of EuroSys*, pages 1–16, 2019.

[42] The Linux kernel development community. BPF Kernel Functions (kfuncs). https://docs.kernel.org/bpf/kfuncs.html.

[43] The Linux kernel development community. XDP RX Metadata. https://docs.kernel.org/networking/xdp-rx-metadata.html.

[44] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of ACM SIGCOMM*, pages 514–528, 2020.

[45] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. HPCC: High Precision Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 44–58. 2019.

[46] Dana Lu, Boxuan Tang, Michael Paper, and Marios Kogias. Towards Functional Verification of eBPF Programs. In *Proceedings of ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 37–43, 2024.

[47] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A Microkernel Approach to Host Networking. In *Proceedings of ACM SOSP*, pages 399–413, 2019.

[48] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, et al. From Luna to Solar: the Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *Proceedings of ACM SIGCOMM 2022 Conference*, pages 753–766, 2022.

[49] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. *ACM SIGCOMM Computer Communication Review*, 45(4):537–550, 2015.

[50] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of ACM SIGCOMM*, pages 221–235, 2018.

[51] Multipath TCP community. Multipath TCP. https://www.multipath-tcp.org/.

[52] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of ACM SIGCOMM*, pages 30–43, 2018.

[53] Atsuya Osaki, Manuel Poisson, Seiki Makino, Ryusei Shiiba, Kensuke Fukuda, Tadashi Okoshi, and Jin Nakazawa. Dynamic Fixed-point Values in eBPF: a Case for Fully In-kernel Anomaly Detection. In *Proceedings of the Asian Internet Engineering Conference 2024*, pages 46–54, 2024.

[54] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of USENIX NSDI*, pages 361–378, 2019.

[55] John Ousterhout. A Linux Kernel Implementation of the Homa Transport Protocol. In *Proceedings of USENIX ATC*, pages 99–115, 2021.

[56] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed Kernel Synchronization Primitives. In *Proceedings of USENIX OSDI*, pages 667–682, 2022.

[57] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[58] The IO Visor Project. eXpress Data Path (XDP). https://www.iovisor.org/technology/xdp.

[59] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. SPRIGHT: Extracting the Server From Serverless Computing! High-Performance eBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of ACM SIGCOMM*, pages 780–794, 2022.

[60] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proceedings of ACM SIGCOMM*, page 691–706, 2024.

[61] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S Berger, James C Hoe, Aurojit Panda, and Justine Sherry. We Need Kernel Interposition over the Network Dataplane. In *Proceedings of ACM HotOS*, pages 152–158, 2021.

[62] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable Traffic Shaping at End Hosts. In *Proceedings of ACM SIGCOMM*, pages 404–417, 2017.

[63] The sched_ext authors. Sched_ext Schedulers and Tools. https://github.com/sched-ext/scx.

[64] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A Cloud-Scale Characterization of Remote Procedure Calls. In *Proceedings of ACM SOSP*, pages 498–514, 2023.

[65] Margo I Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. *ACM SIGOPS Operating Systems Review*, 30(213-228):10–1145, 1996.

[66] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queueing using Deficit Round Robin. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, 1995.

[67] Alexei Starovoitov. bpf: Introduce BPF timers. https://lwn.net/Articles/862136/.

[68] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *Proceedings of USENIX ATC*, pages 431–444, 2017.

[69] Hao Sun and Zhendong Su. Validating the eBPF Verifier via State Embedding. In *Proceedings of USENIX OSDI*, pages 615–628, 2024.

[70] Viet-Hoang Tran and Olivier Bonaventure. Making the Linux TCP Stack more Extensible with eBPF. In *Proc. of the Netdev 0x13, Technical Conference on Linux Networking*, 2019.

[71] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch Dataplane Ten Years Later. In *Proceedings of ACM SIGCOMM*, pages 245–257, 2021.

[72] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-Aware Datacenter TCP (D^2TCP). *ACM SIGCOMM CCR*, 42(4):115–126, 2012.

[73] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the Verifier: eBPF Range Analysis Verification. In *International Conference on Computer Aided Verification*, pages 226–251. Springer, 2023.

[74] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards Domain-Specific Network Transport for Distributed DNN Training. In *Proceedings of USENIX NSDI*, pages 1421–1443, 2024.

[75] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. *ACM SIGCOMM CCR*, 41(4):50–61, 2011.

[76] Qiongwen Xu, Michael D Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. Synthesizing Safe and Efficient Kernel Extensions for Packet Processing. In *Proceedings of ACM SIGCOMM*, pages 50–64, 2021.

[77] Sepehr Abbasi Zadeh, Ali Munir, Mahmoud Mohamed Bahnasy, Shiva Ketabi, and Yashar Ganjali. On Augmenting TCP/IP Stack via eBPF. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, pages 15–20, 2023.

[78] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proceedings of ACM SOSP*, pages 195–211, 2021.

[79] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of USENIX OSDI*, pages 375–393, 2022.

[80] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating Distributed Protocols with eBPF. In *Proceedings of USENIX NSDI*, pages 1391–1407, 2023.

[81] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *Proceedings of USENIX NSDI*, 2024.

[82] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of ACM SIGCOMM*, page 523–536, 2015.

APPENDIX

A Implementing More Transports

In this section, we extend our case studies beyond Homa and TCP, exploring the implementation of other transport protocols (specifically, their host logic) under eTran. We focus on congestion control (CC) while loss recovery and segmentation/reassembly are omitted as they are less important.

Swift [44] is a representative RTT-based transport protocol. It adjusts the congestion window using RTT measurements and distinguishes fabric congestion from host congestion. Swift's CC algorithm can be implemented entirely in XDP. Swift typically uses window control and switches to rate-based pacing when the congestion window falls below 1. This can be implemented with two maps of `BPF_MAP_TYPE_PKT_QUEUE`. Swift requires NIC hardware timestamp, which can be provided via XDP RX metadata [43].

HPCC [45] is a transport protocol that utilizes in-network telemetry (INT) to gather precise load information for CC. The CC algorithm on the sender side of HPCC can be implemented using XDP. Additionally, HPCC employs rate-based pacing with window boundaries, which can be implemented similarly. Specifically, HPCC incorporates a flow scheduler that scans all flows to assign credits. Only flows that have accumulated sufficient credits and are within the window limit can be transmitted. This flow scheduler can be implemented using the dedicated kthread of the BPF timer.

DCQCN [82] is another ECN-based transport protocol that integrates with QCN. On the sender side, DCQCN controls the sending rate using a per-flow timer and a byte counter. The byte counter is updated in XDP, while the per-flow timer can be implemented using an array of BPF timers. Each connection tracks the index of its corresponding BPF timer, similar to how the CC state index is managed in DCTCP. DCQCN enforces rate control entirely based on the rate, which can be implemented similarly to DCTCP.

NDP [30] is a receiver-driven transport protocol that incorporates packet trimming. The receiver-driven logic and credit-based pacing can be implemented similarly to Homa. The main difference is that NDP does not prioritize short messages like Homa. Instead, it lets all flows share bandwidth fairly by default.

B RBtree Operations

The fundamental approach involves utilizing a single red-black tree to manage two distinct trees. We assume each machine is assigned a unique *peerid*, and each RPC is identified by a unique *rpcid*, which consists of a five-tuple. The first tree, termed the *RPC Tree*, maintains the states of all RPCs. The second tree, referred to as the *Peer Tree*, tracks the highest priority RPC from each peer. In the *RPC Tree*, the key for lookups is defined as (treeid = 0, peerid, bytes_remaining, rpcid). In the *Peer Tree*, the key for lookups is defined as (treeid = 1, bytes_remaining, peerid). Keys are compared from left to right. All operations are performed using three APIs: `bpf_rbtree_add()`, `bpf_rbtree_remove()`, and `bpf_rbtree_lower_bound()`, which have the time complexity of $O(\log N)$.

Operations in XDP. When XDP receives a new RPC that needs to be scheduled, it creates an object and inserts it into the *RPC Tree* and takes a snapshot of the value of bytes_remaining in the RPC state for future search. Subsequently, XDP tries to adjust the *Peer Tree*. It starts by searching the *RPC Tree* using the key (treeid = 0, peerid = our_peerid, bytes_remaining = 0, rpcid = 0).

1. If the search results in (treeid $\neq$ 0 or peerid $\neq$ our_peer or rpcid $\neq$ our_rpcid), it indicates that we are not the highest priority RPC for this peer, and no further action is taken.
2. If the search results in (treeid = 0 and peerid = our_peer and rpcid = our_rpcid), it indicates that we are the highest priority RPC for this peer. In this case, we search the *RPC Tree* with the key (treeid = 0, peerid = our_peer, bytes_remaining = our_remaining, rpcid = our_rpcid + 1) to determine if a previous highest priority RPC exists. If such an RPC is found, it is removed from the *Peer Tree*. The new RPC is then inserted into the *Peer Tree*, and its status is marked as present in the *RPC Tree*.

When a new batch of packets arrives for this RPC, a search is performed in the *RPC Tree* using the key (treeid = 0, peerid = our_peer, bytes_remaining = snapshot_bytes_remaining, rpcid = our_rpcid). If no matching object is found, it indicates that the RPC has fully used granted, and no action is needed. Otherwise, it is removed from the *RPC Tree*, its bytes_remaining value is updated, and it is reinserted into the *RPC Tree*. Then, we adjust the *Peer Tree*:

1. If the RPC is already present in the *Peer Tree*, indicating it is still the highest priority RPC from this peer, the key (treeid = 1, peerid = our_peer, bytes_remaining = snapshot_bytes_remaining, rpcid = our_rpcid) is used to search for it in the *Peer Tree*. Then, the object in the *Peer Tree* is removed and reinserted.
2. If the RPC is not present in the *Peer Tree*, the case is treated as a new RPC state.

Operations in XDP_GEN. In XDP_GEN, the key (treeid = 1, bytes_remaining = last_min, peerid = last_peerid + 1) is used to search the *Peer Tree*, where *last_min* and *last_peerid* represent the last minimum bytes_remaining and the last peerid used, respectively. They both start from zero. If the RPC has completed its grant, it is removed from the *Peer Tree*, and the resulting information is then used to search and remove the RPC from the *RPC Tree*.

```
/* Congestion Control */
/* Called at XDP_GEN */
void *bpf_dequeue_ack(struct bpf_map *map);
struct bpf_rb_node *bpf_dequeue_candidate(struct bpf_rb_root *root, bool (less)(struct
    bpf_rb_node *a, const struct bpf_rb_node *b));
/* Called at XDP */
int bpf_update_wnd(struct bpf_map *map, void *key, void *value);
int bpf_update_rate(struct bpf_map *map, void *key, void *value);
int bpf_enqueue_ack(struct bpf_map *map, void *ack_info);
int bpf_enqueue_candidate(struct bpf_rb_root *root, struct bpf_rb_node *n, bool
    (less)(struct bpf_rb_node *a, const struct bpf_rb_node *b);
/* Called at control path */
int cp_read_cc(int map_fd, void *key, void *value);
int cp_update_cc(int map_fd, void *key, void *value);

/* Pacing */
/* Called at XDP_EGRESS */
int bpf_tx_pacing_wnd(struct bpf_map *map, void *key);
int bpf_tx_pacing_rate(struct bpf_map *map, void *key);
struct xdp_frame *bpf_fetch_pkt(struct bpf_map *map, void *key);
int bpf_send_pkt(struct bpf_map *map, int ifindex, __u64 flags);
int bpf_flush_pkt(void);

/* Reliability */
/* Called at XDP */
int bpf_detect_loss(int (cb)(struct xdp_md *ctx, void *value));
/* Called at control path */
int cp_detect_loss(int (cb)(void *cc_ebpf, void *cc_cp);
```

Listing 1: Transport component hooks and corresponding APIs.

C Development Model

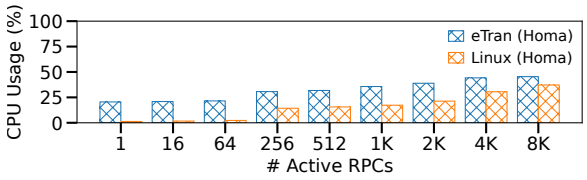
eTran envisions a flexible development model targeting different levels of developers. For transport developers familiar with eBPF programming and transport implementation such as timers and loss recovery, eTran provides a set of low-level APIs for them, as shown in Listing 1. The transport developers can quickly prototype customized transports by directly leveraging these APIs or based on our implementation of Homa and TCP (§4) and then safely deploy them in the Linux kernel. For application developers who focus on user-space application development, they can specify which transport (among eTran-supported ones) to use for their applications. For site reliability engineers and networking administrators, eTran further enables them to conveniently fine-tune the parameters or implementation (thus the performance) of these transports.

We note that our development model is conditioned on the new eBPF hooks and maps in eTran being upstreamed into the Linux kernel. Although this is partially out of the scope of our eTran research, we have tried to minimize changes to the existing kernel eBPF subsystems, thus being friendly to existing eBPF applications as far as we know. Another important factor that determines whether eTran can be upstreamed into the kernel is security, which we elaborate on in §5.

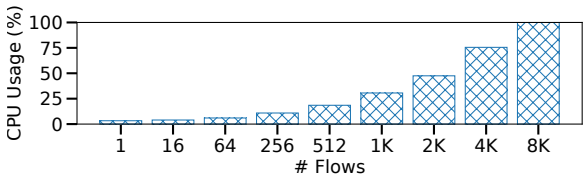
D Additional Design Drill-Down

D.1 CPU utilization of Control Path

We now examine the control path CPU utilization for Homa and TCP. In eTran (Homa), the control path uses a timer thread to check RPC states from the kernel. In eTran (TCP), it runs the CC algorithm and timeout detection. In this experiment, eTran (Homa) performs lookups with a batch size of 16, and eTran (TCP) employs a CC interval of $200\mu s$. We vary the number of RPCs/flows and measure the CPU usage of the control path, the result is shown in Figure 12a and Figure 12b. Both eTran (Homa) and eTran (TCP) exhibit increased CPU usage as the number of RPCs/flows grows. Compared with Linux (Homa), eTran (Homa) incurs higher CPU overhead because it relies on system calls to look up transport states. For eTran (TCP), 8K connections can saturate one CPU core. The control path of eTran (TCP) can be scaled out by partitioning connections across multiple threads.

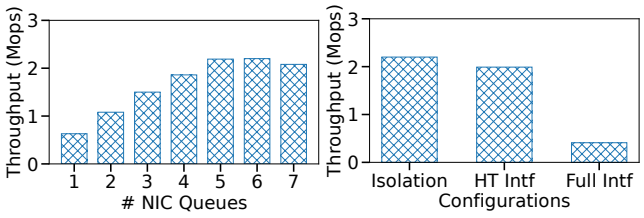


(a) eTran (Homa) vs. Linux (Homa)



(b) eTran (TCP)

Figure 12: CPU usage of eTran (Homa), Linux (Homa), and eTran (TCP), under different numbers of RPCs/flows.



(a) Impact of multi-queues. **(b)** Interrupt interference.

Figure 13: Throughput comparison under different configurations.

D.2 Impact of Multiple Queues and Interrupt Interference

We now examine whether eTran can scale with the NIC queues. We take TCP as an example, as Homa doesn't work well with RSS. We replicate the connection scalability experiment described in §6.2, fixing the number of CPU cores for the application while varying the number of NIC queues. 32K connections are separated across five clients. Figure 13a shows that eTran scales with NIC queues from 1 to 5, but beyond that, throughput slightly declines. This is due to multiple application threads requiring locked access to the Fill ring and Comp ring.

We are also interested in the performance of eTran when the application cores are collocated with the NAPI cores. We conduct the same experiment above with three configurations: 1) separate to different physical CPU cores, 2) separate to different logical CPU cores, and 3) combine on same logical CPU cores. Figure 13b shows the result. The performance of the second configuration is lower than the first mode due to Hyper-Threading (HT) interference. The third configuration exhibits poor performance due to the serve interference between application threads and NAPI [81]. Considering eTran offloads all transport states in eBPF, how to utilize eBPF for CPU scheduling [63] to optimize eTran is an interesting direction. We leave it as future work.