Tux: Efficient Drop-in Networking for Database Systems

Xinjing Zhou
MIT CSAIL
xinjing@mit.edu

Viktor Leis
Technische Universität München
leis@in.tum.de

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

Modern high-performance OLTP systems are increasingly bottlenecked by kernel network stack and the high cost of dispatching requests to database worker threads. While kernel-bypass TCP stacks improve performance, they often sacrifice compatibility and robustness, in addition to leaving performance opportunities on the table due to TCP's byte-stream interface. We present **Tux**, a kernel-bypass networking stack designed for database systems that achieves high performance without giving up compatibility and robustness. Tux addresses these challenges by (1) introducing a message-based transport protocol that decouples reliability from in-order delivery and natively preserves message boundaries without framing or copy overhead inherent in byte-stream interface; (2) providing a flexible pushdown abstraction that lets database engines execute DBMS-specific logic closer to the NIC to avoid context-switch overhead and to exploit message-based interface; and (3) leveraging eBPF/XDP to reuse well-maintained kernel NIC drivers for compatibility and operational ease. We implement Tux in a library called Libtux, offering both zero-change "compatibility" mode and minimal-change "pushdown" mode. Our implementation, Libtux, evaluated on VoltDB, Redis, ScyllaDB, Memcached, and LeanStore, improves throughput up to 2.3x, reduces median and 99th percentile latencies by up to 2.6x and 4.7x, compared to existing kernel-bypass systems with minimal modifications to the evaluated DBMSes.

PVLDB Reference Format:
Xinjing Zhou, Viktor Leis, Xiangyao Yu, and Michael Stonebraker. Tux: Efficient Drop-in Networking for Database Systems. PVLDB, 19(3): 334 - 347, 2025.
doi:10.14778/3778092.3778096

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zxjcarrot/libtux.

1 INTRODUCTION

Database systems have traditionally relied on the standard TCP protocol and OS-kernel stacks for both client-server and inter-DBMS communication. However, recent research [39, 66] has shown that the OS network stack and kernel context switch overhead for dispatching tasks are now the primary bottlenecks in a high-performance production-grade OLTP system, contributing to up to 68% of the total CPU overhead. On the other hand, the rapid improvements in network hardware over the last decade sharply contrast with the well-known stagnant CPU performance. For example, AWS EC2 instances have experienced a 20× increase in network bandwidth over the past decade [3], with Terabit Ethernet [24] on the horizon. The median round-trip latency between two servers in a rack with a commodity network switch and network interface card (NIC) can be as low as single-digit microseconds when excluding OS stacks [6, 43]. The shifting bottlenecks in database systems and the growing imbalance between network hardware performance and CPU performance motivate a rethinking of communication protocols and OS stacks, particularly as the industry moves toward disaggregated architectures in the cloud where components are connected over networks.

A well-known method for overcoming inefficiencies in the OS network stack is kernel bypass. This method typically uses user-space NIC drivers, such as Data Plane Development Kit [5] (DPDK), together with user-space TCP stacks [9, 40, 44, 64, 68]. Some user-space stacks [9, 26, 64, 68] further reduce kernel context-switch overhead by running application logic directly on the network core responsible for packet processing, entirely bypassing the OS scheduler. By eliminating the kernel stack, these solutions can deliver substantial performance gains. However, there are still several unaddressed challenges:

Messaging over Byte-Stream. TCP tightly couples in-order delivery with reliability to expose a byte-stream interface. While most database systems need reliable transport, in-order delivery is not always necessary. Many database operations—especially inter-node communications in disaggregated or replicated systems, such as networked storage reads or consensus-based replication—communicate in a request-response messaging style where in-order delivery is not needed. To operate over a byte-stream interface, a DBMS must encode explicit message boundaries (e.g. via length-prefixing or delimiters) and then parse the incoming byte stream to locate each message boundary. This typically entails buffering partial messages, scanning for headers, and copying payload bytes from TCP buffers, which adds non-trivial CPU overhead.

Integration and Compatibility Challenges. User-space network stacks [9, 26, 64, 68] often require significant changes to application architecture, which limits their adoption in complex database systems. These stacks typically adopt a restricted run-to-completion execution model, where application logic is executed directly on the networking cores to eliminate scheduler and inter-thread communication overheads. This model works well for simple key-value caches like Memcached where responses are generated immediately for each request, but does not generalize to DBMS workloads, where requests may involve complex query execution, interaction with storage engines, or coordination with other nodes

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778096

for distributed transactions and replication. Moreover, many production DBMSes [60, 61, 63] employ dedicated I/O threads that dispatch requests for worker pools for better pipelining. Adapting these architectures to run entirely on the network core would require major rewrites. Deployment challenges further compound these issues. User-space drivers like DPDK require exclusive control of a dedicated NIC, preventing coexistence with standard kernel networking tools. This incompatibility hinders operations such as debugging, monitoring, and cluster management. Together, these restrictions on both execution and deployment significantly raise the barrier to adoption in real-world DBMS deployments. To the best of our knowledge, only ScyllaDB [18, 62] and Yellowbrick [30] support kernel bypass networking, and ScyllaDB do not deploy them in production due to deployment challenges [19].

Robustness Concerns. While running DBMS code on the network core reduces scheduling overhead, this approach becomes fragile when the code is non-cooperative. In OLTP systems, transactions may stall due to long-running queries, I/O delays, or lock contention. A networking core might also become overloaded with request processing. Such unbounded stalls can prevent the network stack from timely processing packets, replenishing NIC buffers in time, resulting in packet drops, starvation, and increased tail latency for short transactions.

In this paper, we show that it is possible to remove the OS-kernel stack bottlenecks while addressing these challenges. We present a new high-performance programmable networking stack designed specifically for database systems, called Tux, which moves the network stack to user space, provides flexible interfaces to co-design network stacks with database systems to exploit message-based reliable communication and avoid kernel context-switch overhead for dispatching tasks, and judiciously leverages modern Linux I/O interfaces for compatibility.

At its core, Tux utilizes a kernel-bypass data path with a reliable transport protocol designed for flexibility and performance. It decouples reliability from strict ordering, offering both a TCP-compatible byte-stream interface for zero-change integration and a native message-based interface to eliminate framing and copy overheads. To tackle context switching overheads, Tux introduces a pushdown abstraction that extends the eBPF paradigm into userspace, allowing DBMS logic (e.g., storage look-ups, transaction processing, command processing) to run directly on networking cores inside Tux stack as a form of callback functions when transport-level data units (stream or message) arrive. Pushdown abstraction can further leverage the message interface for zero-copy data access within callback functions, avoiding overheads. This abstraction enables a wide range of scheduling flexibility to avoid scheduling overhead and data copying. For example, short transactions can be executed directly at the networking cores without incurring context-switching overhead and data copying, while long transactions can be directed to the DBMS's internal pool of worker threads. Furthermore, Tux runs callback functions in a preemptible user-space task runtime that prevents callback functions from monopolizing CPU resources.

For compatibility, Tux leverages well-maintained **kernel-space drivers** to communicate with NIC hardware. This is accomplished via a recent kernel-based bypass path using eBPF/XDP, which avoids the NIC exclusivity requirement. Although using kernel-space drivers inevitably introduces system call (kernel-crossing) and interrupt overhead, we find that the CPU cycles spent in kernel NIC drivers contribute only a small portion of overall DBMS request processing latency. Therefore, using a kernel-based approach for better compatibility represents a worthwhile trade-off. Additionally, Tux aggressively exploits batching and kernel-side polling to amortize kernel-crossing overhead and avoid interrupt overhead, allowing it to match the performance of using user-space drivers.

We implement our design in a library called Libtux that provides two operating modes: compatibility mode and pushdown mode. In compatibility mode, Libtux operates as a shim layer with POSIX interfaces for existing DBMSes that rely on TCP, providing binary compatibility and no DBMS code change. In pushdown mode, Libtux provides an event-driven programming model to minimize porting efforts and facilitate zero-copy messaging.

Our evaluation across VoltDB, ScyllaDB, Redis, Memcached, and LeanStore demonstrates Tux's substantial performance advantages while requiring minimal code modifications. VoltDB on Tux (with zero code changes) achieves 2.3× higher throughput than Linux while reducing end-to-end median and p99 latencies by over 2.2× and 2.8× respectively. Compared to state-of-the-art kernel-bypass stacks, Memcached with Tux's pushdown mode delivers 2.18× higher throughput, 4.19× lower p99 latency, and 72% CPU core savings with just 59 lines of code changes.

To summarize, this paper presents Tux, a high-performance, robust, and DBMS-friendly networking stack that delivers the benefits of user-space I/O without sacrificing compatibility. Our key contributions are:

- **A hybrid kernel-bypass stack for DBMS.** We design and implement Tux, the first database networking stack that leverages eBPF/XDP to retain compatibility with kernel-space NIC drivers while offloading performance-critical paths to user space.
- **A reliable, message-oriented data-path protocol.** Tux provides a reliable data path protocol that supports both: (1) a TCP-compatible byte-stream interface for compatibility of existing DBMSes, and (2) a native message-based interface that eliminates framing/copying overheads for performance.
- **A pushdown abstraction for co-designed execution.** We propose a flexible interface that allows DBMS logic to run directly on networking cores, reducing inter-thread communication and kernel scheduling overhead while enabling new co-design opportunities for cache efficiency and tail latency control.
- **A full implementation and extensive evaluation.** We implement Tux as a userspace library and evaluate it across five DBMSes. Tux improves throughput by up to 2.3×, reduces tail latency by up to 4.2×, and lowers CPU usage by over 70%, with minimal changes to the DBMS engine.

2 BACKGROUND AND MOTIVATION

This section covers the necessary background and motivation, including DBMS networking requirements, modern data path I/O interfaces (DPDK, io_uring, eBPF/XDP/AF_XDP), and kernel-bypass designs. We explain why existing solutions are insufficient and motivate the need for a better DB-OS co-designed stack.

2.1 DBMS Expectations for Networking

We begin by surveying the transport protocols used in major database systems to understand their core requirements, which are

DBMS	Client⇔DBMS	Node⇔Node	Dedicated I/O Threads
MySQL/PostgreSQL/Aurora	TCP	TCP	No
VoltDB/Cassandra/ScyllaDB/ CockroachDB/MongoDB	TCP	TCP	Yes
Redis	TCP	TCP	Optional
Memcached	TCP/UDP	N/A	No
Aerospike	TCP	TCP/UDP	Yes

Table 1: Transport protocols used in DBMSes and whether dedicated I/O threads are employed.

summarized in Table 1. Our findings show that communication largely falls into two categories—client-to-server and internal inter-node traffic—each with distinct needs.

Client-DBMS Communication. For client-server traffic, mainstream database systems rely on TCP. This is a direct consequence of their wire protocols, which are typically connection-oriented for tasks like authentication and expect the strict ordering of a byte-stream to correctly pipeline requests and responses [2, 13, 16]. TCP's wide support and its reliable, ordered byte-stream interface make it a natural and ubiquitous fit for this role.

Inter-Node Communication. Inter-node communication requirements vary significantly, falling into distinct categories based on the protocols in use. Consensus-based replication protocols like Paxos and Raft are designed to operate over unreliable networks, handling message loss and reordering at the protocol level, thereby imposing minimal requirements on the transport itself. In contrast, many other foundational database protocols depend critically on reliable delivery from the transport to prevent catastrophic failures, such as the permanent deadlock from a lost lock-release message in 2PL or the infamous blocking problem from a lost commit message in 2PC. Crucially, for this second group, the need for reliability does not imply a need for transport-level ordering, as logical sequencing is handled at the protocol level using identifiers like transaction IDs or message arrival order (e.g., Lock Manager) or message types (e.g., Prepare happens before Commit in 2PC). This is distinct from a third pattern: operations that require a strictly ordered byte-stream. One example is log-file-based primary/backup replication in systems like MySQL and PostgreSQL, where a sequence of database changes must be streamed and replayed in original order to ensure replica consistency. This requirement also exist in other tasks like transferring intermediate results in distributed query processing.

This analysis highlights that modern database systems simultaneously require both an ordered byte-stream interface, and an efficient, reliable message-passing interface. This dual need poses a challenge for existing transport protocols, which are often too general-purpose or require specialized hardware. Homa [51, 54], for example, is a recent message-based transport protocol designed for datacenter RPC workloads but lacks a byte-stream interface. Scalable Reliable Datagram (SRD) [59, 69] and RDMA [17] also employ message-based protocols but are tied to proprietary hardware or specific network fabrics. Hence, one of the design goals of Tux is to provide efficient messaging over Ethernet that supports both reliable unordered messaging and ordered byte-stream for easy integration into database systems.

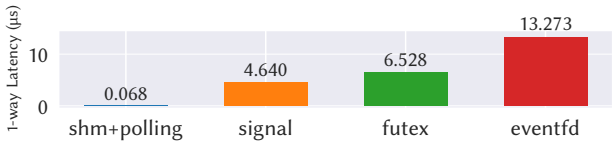


Figure 1: Latency of IPC Mechanisms: The benchmark measures the average one-way latency of ping-ponging 10^6 1-byte messages between two threads on a security-hardened [11] EC2 instance.

2.2 The Cost of Task Dispatching and Kernel Bypass Approaches

Task dispatching—notifying worker threads of packet arrivals—creates significant overhead in high-performance DBMSes. Packets need to traverse hardware interrupts, kernel TCP stack, and OS scheduler wakeups; dedicated I/O threads (Table 1) add further IPC costs. Our microbenchmark (Figure 1) shows kernel-mediated IPC incurs multi-microsecond latency, while busy-polling is too CPU-intensive for multi-threaded OLTP systems. Existing kernel bypass systems use two strategies: co-located [9, 26, 64, 68] (networking and application on same core, eliminating IPC but vulnerable to head-of-line blocking) or separated [31, 40, 44, 50, 53] (flexible scheduling but higher communication overhead).

This trade-off between dispatch overhead and compatibility/flexibility/robustness motivates Tux, which reduces context-switching and scheduling overheads without sacrificing these properties.

Existing kernel bypass solutions thus present a trade-off between minimizing dispatch overhead and maintaining compatibility, flexibility, and robustness. Addressing this challenge motivates the design of Tux, which aims to significantly reduce context-switching and scheduling overheads for network tasks without sacrificing these crucial properties for database systems.

2.3 Modern Data-Path Interfaces

We next describe several modern data-path interfaces available on Linux: DPDK, io_uring, eBPF, and XDP.

User-Space Driver (DPDK). DPDK [5] lets an application take complete control of a NIC and process packets in user space, bypassing the kernel TCP stack. The application runs a loop that busy-polls a receive queue, process the packets (e.g., hands the packet to a user-space TCP stack such as F-stack), and then transmits packets—eliminating interrupts and scheduler wakeups. This yields very low per-packet overhead and predictable latency, at the cost of CPU spent on polling and exclusive NIC ownership.

io_uring. io_uring is a recent asynchronous Linux I/O interface that enables submitting batches of storage or network system calls, thereby amortizing the kernel-crossing overhead. However, it is important to note that each socket operation still traverses the kernel TCP stack, where most of the overhead occurs.

eBPF. The extended Berkeley Packet Filter (eBPF) is a modern Linux technology enabling safe and efficient kernel extensions without modifying the kernel or loading kernel modules. By running sandboxed programs at various kernel hook points—including the networking stack—eBPF allows integration of data-intensive operations directly into the OS networking stack. Recent research has demonstrated its potential for tasks such as database proxying [28],

caching [33], and transaction processing [27, 67], effectively avoiding costly data transfers between user and kernel space. However, eBPF enforces strict safety guarantees: loops must be bounded, programs are limited to access a restricted set of kernel data structures, and floating-point operations are prohibited in the kernel, making it challenging to offload a full-fledged DBMS into kernel space.

Kernel-Space Driver through AF_XDP sockets. Building on eBPF, the eXpress Data Path (XDP) [35] provides a hook that runs eBPF programs as soon as a packet arrives at the network card's driver. From here, an XDP program can decide the packet's fate: drop it, pass it to the normal kernel stack, or—most importantly for Tux —redirect it to a user-space application through a high-speed "express lane" called an **AF_XDP socket**. These sockets use a shared memory region that both the application and the NIC can access directly. This enables true zero-copy packet processing, as data copying between kernel and user space can be elided. Tux leverages this mechanism to bypass the kernel stack and quickly get packets to its user-space data path.

2.4 The Cost of Using Kernel-space NIC Drivers.

While using kernel-space NIC drivers offers superior compatibility, it does introduce kernel-crossing overhead for NIC operations. To quantify this overhead, we perform a micro-benchmark by running Redis on F-Stack [9] atop DPDK—which employs a run-to-completion execution model—providing an upper bound on achievable performance. The following table breaks down the average latency (excluding polling/queuing time) of Redis serving GET requests at a throughput of 300K/s:

Component	Time (μs)	Ratio (%)
NIC Driver Tx/Rx	0.27	7.2%
Socket Send/Recv + TCP	1.51	41.0%
Redis GET	1.91	51.8%
Total	**3.69**	**100%**

The raw NIC driver accounts for only 7.2% of the total time. A kernel-space driver adds the cost of a system call. On a security-hardened [11] `c5n.xlarge` AWS instance, a minimal system call takes about 390ns. This would add roughly 390ns for receiving and 185ns for transmitting, potentially increasing the driver-related portion of latency to around 20% of the total for this simple Redis workload. However, this analysis is conservative. For a more complex database operation taking 20µs, this same system call overhead would drop to just 2.6% of the total time. This reasonable performance trade-off motivates Tux to use kernel-space NIC drivers, gaining compatibility and ease of deployment for a modest cost.

3 Tux DESIGN

Goals. Tux is designed to address the growing mismatch between modern high-speed networks and traditional kernel-based network stacks in high-performance database systems with two goals:

- **G1: Preserve compatibility & high performance.** Tux must support unmodified DBMSes while avoiding the need for kernel modules or exclusive NIC access. At the same time, it must deliver performance comparable to specialized DPDK-based solutions.

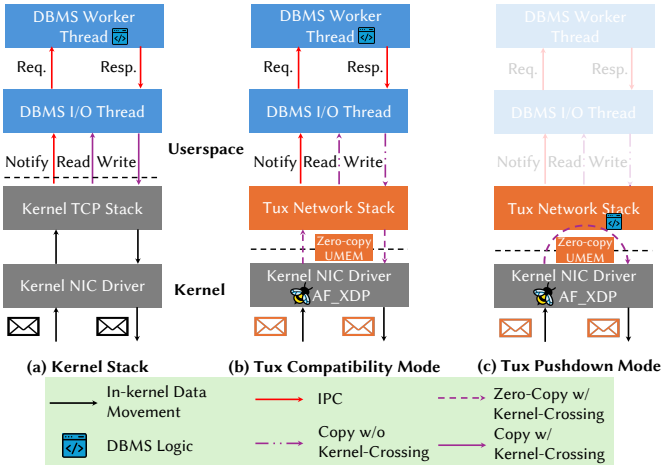


Figure 2: How Request/Response Arrives/Leaves at/from DBMS Worker Thread: (a) Kernel Stack. (b) Tux Compatibility Mode speeds up the network stack with specialized messaging protocol and efficient compatible kernel-based bypass path. (c) Tux Pushdown Mode speeds up network stack and avoids IPC overhead by pushing performance-critical DBMS logic onto networking cores without requiring an overhaul to DBMS's architecture.

- **G2: Enable even better performance via co-design and programmable network stack.** Tux stack should expose lightweight and flexible programming interfaces for DBMS to exploit further performance gains at modest engineering effort.

Overview. These goals shape the overall design of Tux, shown in Figure 2 (b) and Figure 2 (c). To achieve **G1**, Tux runs as a user-space library with POSIX interfaces under the DBMS. When a kernel TCP connection is established, Tux transparently replaces it with a user-space data path that bypasses the kernel TCP/IP and socket layers, without requiring application changes or NIC ownership. The data path uses the reliable message-oriented Tux protocol (Section 3.1), which exposes a native message interface and can emulate byte-stream delivery for compatibility. The Tux protocol is implemented in Tux stack (Section 3.2) on top of recent kernel-based bypass path using eBPF/XDP for accessing NIC driver. Tux stack includes several optimizations (Section 4) to minimize kernel crossing for using the kernel space NIC drivers.

Pushdown. To achieve **G2**, Tux introduces pushdown (Section 3.3): a programming abstraction that lets DBMS logic run directly on networking cores in the form of callbacks, avoiding kernel scheduler and IPC overhead. Pushdown elevates the unit of execution from raw packets to transport-level messages or byte-streams, so DBMS developers can implement request handling at the same granularity as DBMS engine. Conceptually, pushdown extends the eBPF/XDP paradigm into user space: it provides programmable hooks on the data path, but with the full flexibility of user space rather than kernel restrictions. Internally, pushdown is realized via a two-stage pipeline: an in-kernel eBPF/XDP program that quickly steers database traffic through a zero-copy packet buffer to user space, and then Tux executes callbacks inside a preemptible runtime that ensures robustness and fairness, even for long-running logic. Furthermore, callback can tap into the native message interface of Tux protocol to avoid copying and parsing.

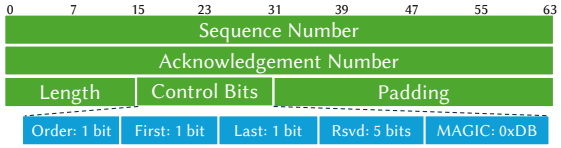


Figure 3: Tux Packet Header

3.1 Tux Data Path Protocol

The Tux data path uses a reliable specialized transport mechanism designed specifically for database workloads within the Tux stack. A key design principle is the decoupling of reliability from strict in-order delivery, inspired by SCTP protocol [22]. Unlike stream-based TCP, Tux inherently operates on discrete messages, aiming to eliminate application-level framing and parsing overhead.

Protocol Mechanics: The structure of a Tux packet, shown in Figure 3, serves the following purposes:

- **Identification and Acknowledgment:** Each packet carries a monotonically increasing sequence number for unique identification, duplicate detection, and reordering, and an acknowledgment number confirming receipt of packets from the peer.
- **Ordering Control:** Three control bits—`order`, `first`, and `last`—control message ordering and fragmentation. Setting the `order` bit enforces in-order delivery on sequence numbers, suitable for stream semantics. Clearing it permits out-of-order delivery.
- **Fragmentation Handling:** For message spanning multiple packets, the `first` and `last` bits act as delimiters. The sender ensures consecutive sequence numbers for packets of the same fragmented message, simplifying reassembly at the receiver.

This small yet flexible header design enables native support for message and optionally stream semantics, delegating the specifics of data segmentation to upper layers.

Message Semantics. For communication patterns where message boundaries are essential but strict ordering is not, Tux provides native message semantics. The sender transmits packets with the `order` bit cleared. The receiver buffers incoming packets, reassembling a complete message (using `first`/`last` bits if fragmented) before delivering it. Delivery can occur via POSIX `recvmsg` or, in pushdown mode (Section 3.3), directly to a callback.

Stream Semantics. Tux can emulate an ordered byte stream for compatibility with TCP. To achieve this, the sender divides the continuous data stream into discrete Tux packets, setting the `order` bit in each. The receiver reassembles ordered packets sequentially using sequence numbers and delivers them through standard POSIX interfaces (e.g., `read`/`recv`) for compatibility.

Delivery. The protocol enforces specific delivery rules based on the ordering requirements. Ordered packets (`order`=1) are delivered to the application strictly in sequence, only after all preceding packets have been consumed. Unordered packets (`order`=0) containing a complete message can be delivered immediately upon arrival. If a message spans multiple packets, delivery is deferred until all constituent packets have arrived.

Reliability. Tux ensures reliable data transfer through sequence numbers and acknowledgments. Sequence numbers allow detection of duplicates and management of reordering. The sender retains packets until the receiver acknowledges successful receipt up to a specific sequence number (X), at which point packets with sequence numbers ≤ X are discarded. A straightforward retransmission mechanism, based on timers and duplicate acknowledgments (triggered after three duplicate ACKs), handles packet loss.

Packet Buffer Reclamation. Efficient memory management requires careful reclamation of packet buffers. A buffer holding a packet is reclaimed only after the application has fully consumed its contents. In in-order scenarios, reclamation occurs immediately upon consumption. However, in mixed or out-of-order scenarios, reclamation of a consumed out-of-order packet may be postponed until all packets with lower sequence numbers have also been consumed and reclaimed, ensuring reliable duplicate detection.

UDP Encapsulation. Each Tux packet is encapsulated in a UDP packet which allows the reuse of checksum hardware offloads for UDP on modern NICs. Hence, the Tux header does not contain a checksum. When a Tux connection replaces a kernel TCP connection, it inherits the TCP connection's port assignments and next-hop routing information, allowing reuse of kernel functionalities.

Comparison to SCTP. While both Tux and SCTP offer reliable, message-oriented transport beyond TCP, Tux is not general-purpose and is specialized for database systems for both compatibility and performance. SCTP is a general-purpose message protocol mainly used in mobile networks with features [23] like multi-homing, multi-streaming, and a complex, extensible chunk-based packet format that are unnecessary in database workloads. In contrast, Tux uses a simpler header structure tailored for its dual stream/message interfaces and relies on UDP encapsulation for common checksum hardware offloads, which are less frequently available for SCTP. Thus, Tux prioritizes performance and tighter integration with database systems.

3.2 Tux Components

Figure 4 presents the internal of Tux stack which consists of a compatibility layer, a shared-nothing architecture for managing concurrency and memory, the Tux runtime for running pushed-down callback functions, and kernel-space NIC drivers, as discussed below.

Compatibility (Shim) Layer. Since most DBMSes run on top of the POSIX interface, Tux implements a POSIX shim layer on top of Tux stack to support running unmodified DBMSes. Tux intercepts `glibc` library (via `LD_PRELOAD` or linking) functions related to networking (e.g., `connect`/`send`/`recv`/`epoll`) and replaces them with Tux implementations to leverage the Tux data path. Each Tux connection is uniquely identified by the file descriptor that represents an established kernel TCP connection. The interposition for a kernel TCP connection is done once it is established and the Tux handshake confirms that both ends are running the Tux stack.

Connection Establishment. Since a Tux connection is attached to a TCP connection and running custom protocols, it needs to perform a handshake to confirm that both ends are running the Tux stack. The handshake is initiated when a kernel TCP connection is established. The handshake process involves sending a special Tux packet to the remote end. Only when both ends receive a handshake packet can the Tux connection be considered established. If a handshake packet is not received within a timeout, Tux stack falls back to using kernel-based TCP for compatibility.

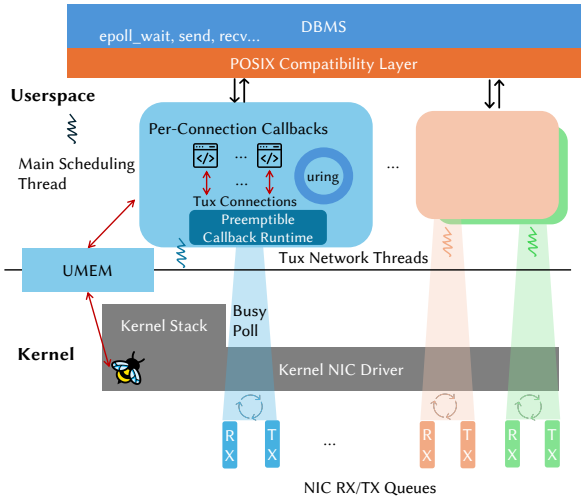


Figure 4: Tux Stack: Each Tux network thread manages a set of NIC queues via a corresponding set of AF_XDP sockets. Due to NIC receive-side scaling, packets of a Tux connection will also be mapped to the same NIC queue and, hence, the same Tux network thread.

Concurrency Management. Tux adopts a shared-nothing architecture to manage concurrency and scalability. Tux uses a set of network threads to process network packets. Each network thread is also responsible for executing callbacks when running in pushdown mode. Each Tux connection is exclusively handled by a single network thread after establishment. Modern NICs employ receive-side scaling (RSS) to deterministically distribute load among multiple NIC queues, each processed by different CPU cores for scalability. This is typically achieved by hashing transport-layer header values (e.g., IP addresses and ports), ensuring that all packets for the same connection are directed to the same NIC queue. Consequently, once a Tux connection is established, its corresponding NIC queue is fixed for the lifetime of the connection, eliminating the need for inter-thread synchronization for connection state management.

Memory Management. To eliminate contention, Tux allocates a user-space memory region per NIC-queue shared between userspace and kernel via the AF_XDP socket APIs [21]. The region is divided into 4KB pages. Each page may be used as a packet buffer for both receiving and transmitting by the NIC hardware. This ensures that all packets belonging to a Tux connection reside on the same UMEM region, avoiding coordination between NIC queues.

3.3 Pushdown Abstraction and Runtime

The pushdown abstraction can be viewed as a user-space extension of the eBPF paradigm, but operating at the transport layer rather than on raw packets. Whereas kernel eBPF provides a safe but restricted environment for filtering and simple packet operations, Tux pushdown exposes a programmable interface on DBMS-relevant transport units: reliable messages and ordered byte streams. This design elevates the programming model from low-level packets to semantically meaningful objects—such as a SQL request, a log page, or a key–value lookup—matching the abstractions DBMS developers actually care about. Unlike kernel eBPF, pushdown executes

```
struct tux_routine {
    char* stack
    bool uninterruptible
    bool interrupted
}

def tux_preemption_signal_handler(signo):
    r = active_routine
    if r == null or r->uninterruptible or r->interrupted:
        return
    r->interrupted = true
    switch back to tux network loop
```

Figure 5: Tux Preemption Signal Handler: Preemption is skipped when a tux routine is in an uninterruptible state.

unrestricted user-space logic with access to the full runtime environment, while the Tux stack manages transport-level complexity such as reliability, fragmentation, and reassembly. This approach combines the efficiency of kernel-bypass with programmability, enabling DBMS developers to attach logic directly to transport events. Currently, Tux supports two types of per-connection callback on arrival of Tux messages:

- **StreamCallback(TuxContext* ctx, void* user_state):** The stream callbacks are executed when the Tux connection has available input data. The callback may use POSIX APIs (e.g., `read/recv/write/send`) to read the data that provides in-order delivery and run arbitrary data-processing logic. Stream callbacks are typically used to implement client-facing communication, such as wire protocol and SQL command processing, where in-order delivery is necessary for compatibility.
- **MessageCallback(TuxContext* ctx, void* user_state, TuxMessage[] msgs):** The message callbacks are executed when a group of unordered Tux messages arrive. Message callback may directly access the message content stored in the UMEM packet buffer without copying it to the application buffer, as the lifetime of messages is the same as the callback execution. Such a primitive can be used to build messaging without the overhead of parsing and copying. This is useful for inter-node communication, where trading compatibility for performance is easier.

Each callback function is associated with a single Tux connection identified by the corresponding TCP connection's file descriptor(fd). Developers may also associate the user state with the callback upon registration. Tux stack provides a runtime for robustly running callback functions for each connection which describe next.

Tux Lightweight Task. A Tux callback is executed inside a lightweight user-space thread, called Tux routine, that runs on the networking core, similar to a go-routine [10]. Each routine maintains its own stack and a small region of memory for storing register values during context switches on preemption. We avoid stackless user-level threads [4] as a stack-based design provides better compatibility with unmodified user code, despite slightly higher context switch overhead [34].

Grouped Execution. Running each Tux callback function in its own routine is inefficient, as it incurs one context switch per invocation and requires a stack per routine. To reduce both overheads, Tux runs multiple callbacks into a single Tux routine. This amortizes the cost of switching and reduces memory usage. When a new group of callbacks is ready and there is no idle or preempted

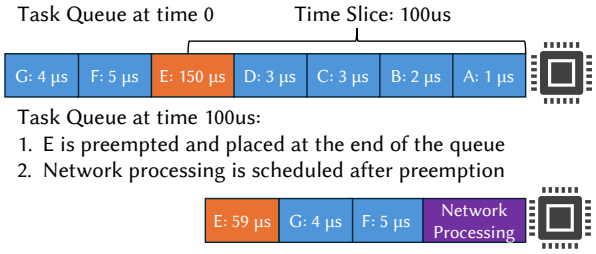


Figure 6: FIFO Scheduling with Timeout: E is preempted after the time slice is exhausted. E is enqueued to give other tasks a chance to run. Network processing is scheduled between callback executions.

routine available, Tux creates a new routine to run them. As a result, the number of active routines is proportional to the number of preempted executions, not the total number of Tux connections.

Scheduling. Tux time-shares between callback execution and network processing. Callbacks are assigned a fixed time slice, while network processing runs without time constraints. Within the time slice, Tux schedules callbacks using FIFO with a timeout. As shown in Figure 6, callbacks are invoked in arrival order from a task queue. If the time slice expires while a callback is still executing, the routine is preempted and moved to the end of the queue, allowing other callbacks to run in the next round. This prevents a long-running callback function from monopolizing CPU resources. To handle blocking I/O, Tux converts common `glibc` calls (e.g., `pread/pwrite`) into asynchronous I/O via `io_uring`, voluntarily yielding control back to the Tux scheduler.

Preemption. Tux uses Linux signals to perform preemption for its wide availability in both cloud VM and bare-metal hardware. The main scheduling thread (Figure 4) checks the run time of each active tux routine and sees if it exceeds the allocated time slice. When a violation happens, it sends a signal to the thread running the routine which is processed by a signal handler, as shown in Figure 5. The handler performs a context switch on the actively running Tux routine and switches back to the network processing context. Because the Linux signal has an overhead of a few microseconds, as shown in Figure 1, we limit the time slice to be at least 100μs to make sure the preemption overhead is not significant. This approach offers a practical balance between minimizing tail latency for short tasks and maintaining high overall throughput [38].

Uninterruptible Region. There are cases where preemption causes problems. For example, when Tux routine is manipulating Tux runtime data structures, preemption might cause the runtime data structures to be in a corrupted state, resulting in crashes. Tux routine leverages a flag (uninterruptible) in memory to denote whether a callback is in an uninterruptible state. Tux runtime leverages such a flag to build critical sections that cannot be interrupted or preempted. As shown in Figure 5, after receiving the preemption signal, Tux checks the flag and aborts the preemption attempt if the uninterruptible flag is set. Tux also makes sure that the preemption handler itself is implemented in a re-entrant fashion.

3.4 DB-Network Stack Co-Design Patterns

Next, we describe several co-design patterns enabled by the proposed push-down abstraction to avoid scheduler overhead. Our key design principle is to optimize away the kernel scheduler overhead for short-running DBMS operations as they are most severely impacted by the cost of context switch and long-running requests.

```
def tux_auto_scale(U_useful: [0, 1]):
    N = num_tux_net_thread
    if N >= num_nic_queues:
        return
    if U_useful > T_scaleup:
        Allocate thread T_N+1
        Rebalance NIC queues among N+1 threads
    else if U_useful < T_scaledown && N > 1:
        Remove thread T_N
        Rebalance NIC queues among N-1 threads
```

Figure 7: Tux Auto Scaling Algorithm.

Complete Pushdown: For workloads dominated by very simple, fast operations, the entire request processing logic can be executed within a callback on the Tux networking cores. Examples include point lookups/updates in key-value stores or caches, and simple storage reads, where request processing time is typically in the tens of microseconds. This pattern minimizes scheduling overhead and maximizes cache efficiency for these latency-critical requests. This pattern works for both callback types.

Selective Pushdown: Many database workloads involve a mix of short-running and long-running requests [29]. For more complex transactions (e.g., TPC-C Stock-Level), range queries, aggregations, or joins, which might run for longer durations or require coordination/parallelism across multiple cores, it is often beneficial to dispatch them to the DBMS's existing worker pool. Developers can implement this by registering a callback that acts as a dispatcher. The callback parses the incoming request, identifies its type or complexity, and decides whether to process it directly for better efficiency or forward it to a worker pool for better tail latency.

3.5 Auto-Scaling Tux

Executing callbacks on the network core consumes CPU cycles. When a single core is overloaded, latency will increase dramatically. Tux introduces an auto-scaling mechanism that reacts to CPU overload to spread the load over multiple network threads. Tux adopts a per-core CPU utilization-based scaling strategy. Specifically, Tux tracks the fraction of CPU cycles spent in user space doing useful work, defined as U_{useful}, which is derived from $1 - U_{polling}$ where $U_{polling}$ is the fraction of CPU cycles spent in kernel drivers. Tux periodically checks the U_{useful} of each network thread in the main scheduling thread. The scaling steps are described in Figure 7. When the U_{useful} estimation exceeds a threshold $T_{scaleup}$, Tux allocates a new network thread and redistributes the NIC queues among the new set of network threads. Due to Tux's share-nothing design (i.e., a NIC queue and all the connections belonging to the queue are exclusively managed by a single network thread), Tux only supports scaling the number of network threads up to the number of supported NIC queues. Similarly, when Tux U_{useful} drops below a threshold $T_{scaledown}$, Tux removes a network thread and re-balances the NIC queues among the new set of network threads.

3.6 Discussion

Deployment. Tux has one limitation in that Tux requires both ends to run the Tux stack and protocol. This might be fine for inter-node communication where DBAs typically have more control over.

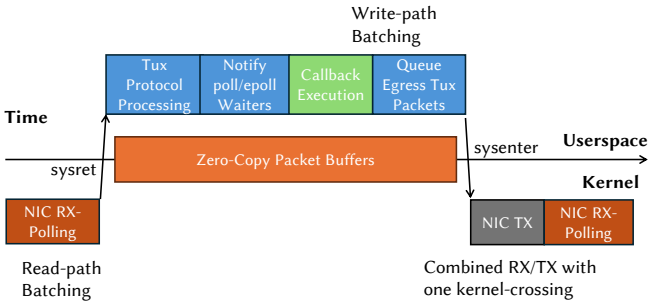


Figure 8: One Iteration of the Processing Loop in a Tux Network Core and Optimizations to Reduce System Call Overhead: Read-path Batching busy-polls in the kernel NIC driver to collect a batch of packets; Write-path Batching prepares a batch of packets before going into the kernel; Combined RX/TX performs packet transmission and RX polling with a single system call.

However, we believe this limitation to be less of a concern as eBPF, on which Tux relies, is getting increasingly widespread [8] in the cloud [7]. Compared to DPDK-based solutions, Tux does not require kernel modules or full NIC control or huge-pages. Furthermore, the Tux compatibility mode allows easy integration with applications to connect to a Tux-enabled DBMS server.

Generality. While this work was motivated by the specific bottlenecks we observed in high-performance OLTP database systems, the design principles are broadly applicable. The core patterns embodied in Tux are well-suited for a wide range of data-intensive, latency-sensitive applications, particularly those that handle a mix of request complexities and benefit from a message-oriented communication. Beyond caches we explored in the paper, *microservices*, *API gateways*, and *HTTP servers* could leverage selective pushdown to handle simple tasks on the network cores while dispatching more complex requests to a worker pool. Even in analytical (OLAP) workloads, where long-running queries are not ideal for pushdown, the underlying high-performance data path can accelerate the data shuffling and transfer of intermediate results in distributed query execution. We leave these extensions as future work.

Security. Tux is designed to be encryption-agnostic, operating at the transport layer underneath userspace libraries (e.g., OpenSSL) that handle TLS encryption. By preserving TCP-compatible stream semantics, Tux supports standard protocols like TLS 1.2/1.3 without modification. The Tux stack neither inspects nor alters encrypted payloads, and no sensitive material such as TLS keys or certificates is exposed outside of the DBMS process. Consequently, Tux provides the same security guarantees as other kernel bypass stacks that focus on accelerating the transport layer.

4 IMPLEMENTATION AND OPTIMIZATIONS

We next describe several key implementation details of the Tux stack that enable robust and efficient kernel-bypass execution. These include concurrency management, safe handling of thread-local storage in user-space task switching, and optimizations for avoiding preemption overhead for short callbacks and optimizations for batching to reduce the cost of using kernel-space drivers.

Managing Concurrency. DBMS worker threads interact with Tux connections through standard POSIX interfaces (e.g., `send/recv`). To safely support concurrent access, Tux use per-connection latch for safety. This design is based on the observation that most connections are typically accessed by a single DBMS thread, making latch contention rare. Additionally, Tux network threads are decoupled from these latches during normal operation. Instead, each Tux connection uses two lock-free ring buffers: one for queuing incoming packets to be consumed by the DBMS, and another for outgoing packets. This design minimizes contention between DBMS threads and network threads.

Handling Thread-Local Storage. Modern DBMSes and memory allocators rely heavily on thread-local storage (TLS) for scalable synchronization. However, when a Tux routine is preempted mid-execution, it may leave TLS variables in an inconsistent state. Since all routines share the same OS thread, switching to another routine can potentially corrupt TLS variable state. On x86, TLS is accessed via a segment register whose base address is stored in the `fsbase` (or `gsbase`) register. To support arbitrary callback functions that use TLS, each Tux routine maintains a dedicated region of memory for TLS storage. During context switching, Tux saves and restores the segment base register using `rdfsbase`/`wrfsbase` instructions, ensuring TLS isolation between routines.

Avoiding Preemption Overhead. To avoid unnecessary preemption, the Tux runtime tracks the execution time of each routine. If a routine has consumed more than 90% of its time slice after finishing a callback, it voluntarily yields control to the scheduler. This optimization allows routines running short callbacks to avoid being preempted entirely, reducing preemption overhead while maintaining fairness.

Optimizations for using Kernel Drivers. As explained in Section 2.3, using kernel-space drivers introduces kernel-crossing overhead. To mitigate this, Tux's network processing loop, shown in Figure 8, is designed to maximize batching and minimize system calls. The loop begins with ***Read-path Batching***. Instead of making a system call per packet, Tux uses a single call to leverage kernel-side polling, which disables interrupts and receives an entire batch of packets into the shared UMEM buffer. After protocol processing, Tux notifies any DBMS threads waiting on `poll/epoll` and then proceeds to execute callbacks, allowing for better pipelining between the network stack and DBMS workers. For ***Write-path Batching***, outbound packets generated by both DBMS threads and callbacks during this process are queued. At the end of the loop, all queued packets are sent in a single batch. This pipeline enables an optimization where the system call to transmit the outbound batch is combined with the call to poll for the next inbound batch.

With these optimizations, Tux effectively pays only one kernel-crossing overhead on the critical path for an entire request-response cycle, drastically amortizing the per-packet system call cost.

5 EVALUATION

In this section, we conduct extensive experiments to answer the following questions:

- How does Tux perform on real-world high-performance database systems? (Section 5.2)
- How much performance benefit does the push-down abstraction provide for database systems? (Section 5.3)

Kernel Bypass Stack	Redis	VoltDB	Memcached
F-stack	678	1,300	–
Demikernel	1,958	–	–
TAS	0	–	0
Tux (Compatibility)	0	0	0
Tux (Pushdown)	63	–	59

Table 2: Code changes for porting DBMSes to kernel bypass stacks.

- What are the performance benefits of using native unordered message-based communication for database systems? (Sections 5.3 and 5.5)
- What are the benefits of tighter co-design between DBMS work scheduling and Tux stack on transaction throughput and tail-latency? (Section 5.4)

5.1 Baselines and Environment

We compare Tux mainly against four kernel-bypass systems: TAS [44] at commit `d3926b`, F-stack [9] at commit `bdd7bed`, and Demikernel [64] at commit `db4b938`, ScyllaDB v6.1.0 with userspace TCP stack running on DPDK v23.07.0. TAS runs a user-space TCP stack in a process as a microkernel service. DBMS connects to the TAS process via shared memory. F-stack and Demikernel are single-threaded user-space TCP stack that adopts a run-to-completion execution model and runs directly in the same DBMS process. We configure Tux runtime to use a preemption time slice of 100μs.

All experiments are conducted on two `c5n.4xlarge` instances on AWS. Each instance has 16 vCPUs virtualized from an Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz and 42 GB of memory. The instances are closely placed in a single availability zone and via the *cluster* placement group policy [15]. The system uses an unmodified Linux kernel (v6.8.0) running Ubuntu 24.04. The instances are equipped with Elastic Network Adapter running v2.13.2g driver with 25Gb/s bandwidth. We use a Maximum Transmission Unit (MTU) of 3498 bytes supported by ENA [12]. The measured median round-trip time between two instances is about 34 μs with DPDK.

5.2 Comparison on Real-World Systems

We start by evaluating Tux against kernel-bypass baselines on four real-world database systems and a cache system. By default, we use the TAS kernel-bypass stack on the client instance to communicate with the server. This allows a fair comparison as both the server and client sides are running kernel-bypass stacks. By default, we use stream-based callback for Tux pushdown mode as the benchmark focuses on client-DBMS communication, where TCP-style byte-stream compatibility is required. We evaluate the message interface in the next sections. We list code changes required for porting to use various kernel bypass stacks in Table 2. F-stack and Demikernel both require orders of magnitude more code changes compared to Tux due to new APIs and architectural constraints.

5.2.1 ***Redis*** We use `memtier_benchmark` [14] to generate GET requests with a uniform key access pattern to the Redis server storing 1M 1KB-sized records. We increase the load by increasing the number of connections to 128, each with a pipeline depth of 1.

The results are shown in Figure 9. Tux-Compat delivers the highest achievable throughput, outperforming Linux/F-stack/TAS by 2.47×/1.64×/1.27×. While Tux-Compat mode achieves higher

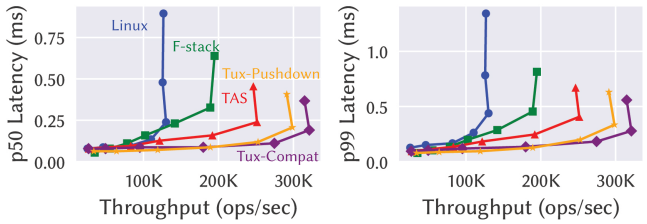


Figure 9: Throughput-Latency on Redis: Tux variations deliver the highest throughput and lowest latency. (Workload: 100% GET, Database: 1M 1KB-sized records).

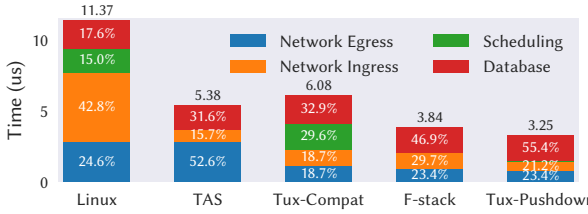


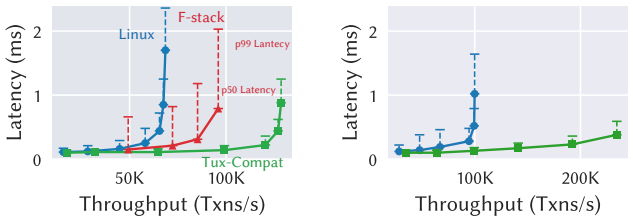
Figure 10: Server-side Latency Breakdown of Redis GET Request on Kernel Bypass Stacks. (Throughput = 50K ops/s on 1M 1KB-sized records, excluding polling, queueing, and off-CPU time).

throughput than Tux-Pushdown, it requires two cores, which pipelines Redis command processing and Tux protocol processing compared to just one core in Tux-Pushdown. When operating at 90% of Linux achievable throughput, Tux-pushdown improves P50 latency by 4.1×/2.63×/1.74×/1.23× and P99 latency by 5.25×/2.45×/1.87×/1.3×, compared to Linux/F-stack/TAS/Tux-Compat, with only 59 lines of modifications to Redis. We omitted results for Demikernel as it does not scale beyond 70K/s throughput due to bottleneck in its co-routine scheduler.

Latency Breakdown. To understand each kernel bypass stack in more detail, Figure 10 presents the server-side latency breakdown at a relatively low load (throughput=50K/s). First, all kernel-bypass systems have lower latency compared to the Linux stack, as expected. The reductions are from more efficient network stacks and reduction in kernel-boundary crossings. TAS has an slightly better overall latency compared to Tux-Compat because it uses busy-polling on shared memory which has lower latency than Tux-Compat's epoll-based communication. Tux-Pushdown offers the lowest overall server-side latency for serving a request, despite using kernel drivers, outperforming F-stack by 18%. We attribute this improvement to the optimizations Tux employs for using kernel drivers, efficient implementation, and protocol simplicity.

5.2.2 ***VoltDB*** We evaluate Tux's compatibility mode on VoltDB (configured with 8 worker threads) against Linux and F-stack. TAS was excluded as it does not support the local loopback TCP connections used internally by VoltDB. Integrating the single-threaded F-stack was invasive, requiring 1300 lines of code change [25] and limiting its evaluation to a single I/O thread. In contrast, Tux works with zero modifications.

As shown in Figure 11, Tux's CPU-efficient protocol and stack deliver significant gains. With a single I/O thread, Tux improves throughput by up to 1.87× and reduces p99 latency by up to 4.72×



(a) 1 VoltDB I/O Thread. **(b)** 4 VoltDB I/O Threads.

Figure 11: Throughput-Latency of YCSB-C Transactions on VoltDB. Marker=p50; whisker=p99. Tux scales to 2.3× higher transaction throughput and significantly improves median and p99 latency. (F-stack results are excluded for 4 I/O threads as it does not support multiple I/O threads for VoltDB.)

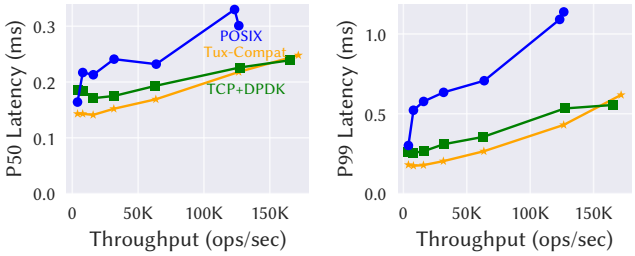


Figure 12: Throughput-Latency on ScyllaDB : Tux-Compat provides end-to-end substantially lower P99 latency and comparable P50 latency compared to ScyllDB's DPDK+TCP stack and POSIX stack (Workload: YCSB-C, Database: 10^6 1KB records, # Tux Net Thread=1).

compared to the baselines. When scaling to 4 I/O threads—a configuration F-stack cannot support—Tux achieves 2.3× higher throughput than Linux, while also reducing median and p99 latency by 2.17× and 2.82× respectively. These results demonstrate that Tux provides substantial throughput and latency improvements on a full-fledged relational OLTP system without any code modification.

5.2.3 ***ScyllaDB*** We evaluate Tux on ScyllaDB—a system engineered from the outset for maximum performance with an optional user-space TCP/IP stack on DPDK. Its shared-nothing, shard-per-core design [20] intentionally co-locates TCP/IP processing with query execution to minimize kernel involvement and synchronization. Because this design executes each query on a single core without latches, Tux 's pushdown mode (which would run DBMS logic on networking cores) is incompatible: it would require safe cross-core access to database objects. We therefore run Libtux in compatibility mode to transparently replace the POSIX stack. We use `cassandra-stress` [1] to generate a YCSB-C workload, configuring ScyllaDB with 8 CPU cores under both the POSIX and native TCP+DPDK stacks. For Tux-Compat, we dedicate 7 cores to request processing and 1 core to the Tux network thread.

As shown in Figure 12, Tux-Compat achieves a similar peak throughput to ScyllaDB's TCP+DPDK stack; both outperform the POSIX stack by 36%. At low to moderate load (<120K ops/s), Tux-Compat delivers 13–23% lower median latency and 20–31% lower P99 latency than the native TCP+DPDK stack. At higher loads, however, the native TCP+DPDK configuration pulls ahead—consistent

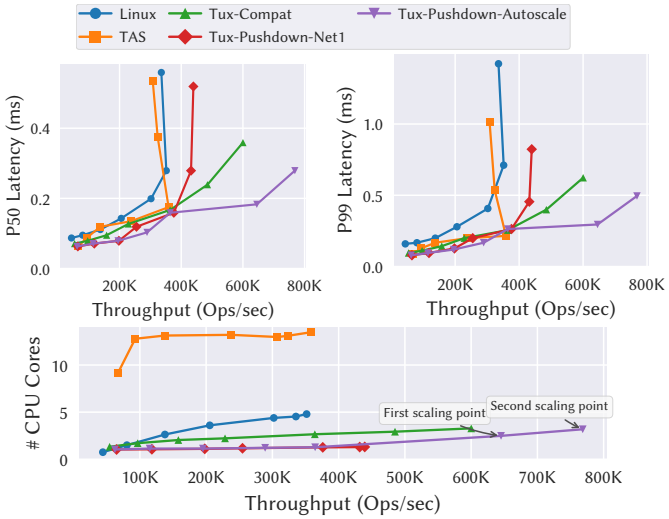


Figure 13: Throughput-Latency on Memcached: Tux delivers highest throughput. Pushdown mode consumes the least CPU resources at a given throughput point. (Workload: YCSB-C with 1KB payload; Tux-Pushdown-Autoscale Configuration: $T_{scaledown} = 0.2$, $T_{scaleup} = 0.9$).

with ScyllaDB's design intent—because packet processing and request handling run on the same cores, avoiding IPC, and scaling more smoothly. In contrast, Tux 's single network thread becomes a bottleneck as load increases, which is reflected in the steeper latency growth for Tux-Compat. The small gap overall is attributable to Tux 's simple protocol and batching optimizations.

5.2.4 ***Memcached*** We evaluate Tux on Memcached, configured with 8 worker threads, comparing it against TAS and Linux, as Demikernel and F-stack are incompatible with its multi-threaded architecture. We test Tux in three configurations: compatibility mode (Tux-Compat), pushdown mode on a single network core (Tux-Pushdown-Net1), and pushdown with autoscaling. Integrating the pushdown mode required only 59 lines of code, a minimal change attributed to the natural fit between Tux's interface and Memcached's event-driven processing model.

The results in Figure 13 show that Tux's latency benefits become more prominent as load increases. While the performance gap between compatibility and pushdown modes is within 25% at low loads, the trade-offs become clear at scale. Tux-Compat can achieve higher peak throughput than Tux-Pushdown-Net1 because it pipelines command processing in worker threads with network processing in the Tux thread. However, with auto-scaling enabled, the pushdown mode outperforms all other configurations, achieving 2.18×/ 2.28×/1.28× achievable throughput compared to Linux/TAS/Tux-Compat, respectively. At 90% of Linux's maximum throughput, this configuration reduces p99 latency by 4.19×.

These latency reductions are accompanied by substantial CPU savings. While Linux requires ≈4.45 cores near its peak, Tux-Pushdown-Autoscale uses only 1.22 cores (a 72.66% reduction). This efficiency stems from utilizing CPU cycles that would be spent polling to instead execute Memcached commands. In contrast, TAS exhibits very high CPU usage (13.04 cores) for a modest latency reduction, a result of its busy-polling IPC mechanism. The Tux auto-scaling

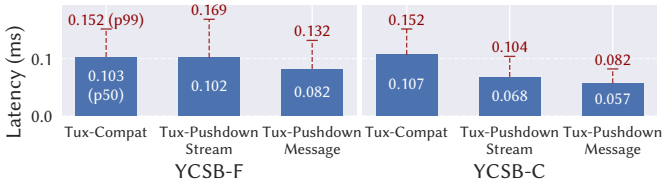


Figure 14: End-to-end Latency of YCSB Transactions on Networked LeanStore: Push-down results in up to 31/25% reduction on p50/p99 latency. Message interface further reduces the p50/p99 latency by up to 22%/25%. (We use a database of 10^6 1 KB-sized records running at 180K txns/second)

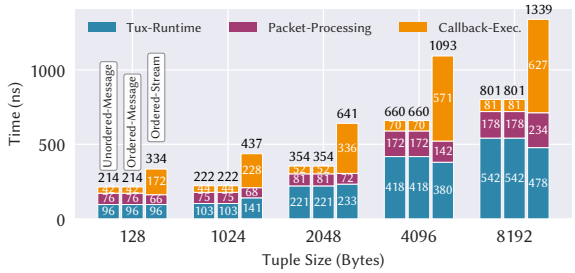


Figure 15: Server-side Per-Request Overhead Breakdown for Various Packet Delivery Approaches: we categorize CPU time spent in Tux-Runtime (Tux-Runtime), network packet processing (Packet-processing), and executing DBMS logic (Callback-Exec.).

mechanism dynamically allocate cores as the offered load increases to maintain low latency while maximizing resource efficiency.

5.3 Benefits of Push-down Execution and Message Interface

To quantify the benefits of pushdown execution and the message interface, we built a transactional server using LeanStore [45] to execute YCSB-C and YCSB-F stored procedures. The baseline server architecture uses a dedicated I/O thread to read from Tux and dispatch requests to a worker thread. In our controlled experiment, we use one of each thread type. The results are shown in Figure 14. The Tux-Compat baseline uses three threads (I/O, worker, and network), enabling pipelining, whereas pushdown mode uses only a single thread for all tasks. Despite losing this pipelining, pushdown with a stream callback still reduces median latency on YCSB-C/YCSB-F by 11% and 31% respectively, due to lower scheduler overhead and cache coherence traffic. Switching to the message-based callback provides a further median latency reduction of 22% and 16% by eliminating byte-stream parsing and data copying. Overall, message-based pushdown improves p99 tail latency by up to 70% compared to the pipelined Tux-Compat baseline.

5.4 Benefits of DB-Network Stack Co-designs

This section demonstrates how co-designing DB transaction scheduling with Tux optimizes the TPC-C workload on LeanStore, which features a mix of 92% short (Order-Status, New-Order, Payment) and 8% long (Stock-Level and Delivery) transactions. We compare three approaches: (1) **No-Pushdown**, a baseline using Tux's compatibility mode to feed two LeanStore worker threads; (2) **Complete-Pushdown**, which executes all transactions in callbacks on a single

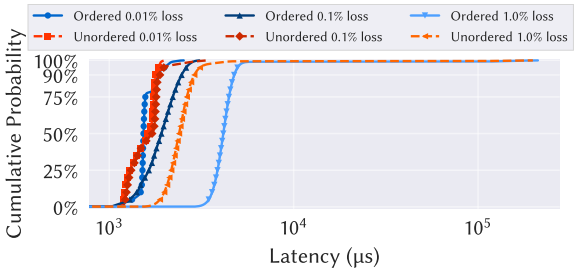


Figure 16: Cumulative One-way Latency Distribution for Transferring 1024-byte Messages under Varying Packet Loss Rate: Unordered transfer allows for lowest latency under loss.

Figure 17: End-to-end Latency of TPC-C on Networked LeanStore: Selective pushdown with 32-message group results in lowest median/p99 latency for short/long transactions. (We use a database of 10 warehouses with an 8GB buffer pool running at ≈ 31,000 transactions/second; We use message-based callback for pushdown mode).

Tux network thread; and (3) **Selective-Pushdown**, where a callback acts as a dispatcher, executing short transactions directly but forwarding long ones to the worker threads. For the latter, we evaluate two variants using message groups of size 1 and 32, as larger groups can improve resource utilization via better pipelining.

The results are summarized in Figure 17 at a throughput of 31,000 transactions/s. The selective pushdown approach with group size of 32 messages provides the best performance for short transactions, reducing median latency by 30% versus No-Pushdown and 57% versus Complete-Pushdown. The p99 latency improvements are even more pronounced, with reductions of 52% and 69%, respectively. Even long transactions benefit, with an 8% latency reduction compared to No-Pushdown, primarily because the callback bypasses the LeanStore I/O thread, reducing communication overhead.

In terms of resource efficiency, selective pushdown with GroupSize=32 achieves 54% higher throughput per core than No-Pushdown, while sacrificing only 15.3% efficiency compared to Complete-Pushdown. This trade-off is justified by the substantial latency improvements

for the dominant, performance-critical short transactions in the TPC-C workload.

5.5 Cost of Ordering and Byte-Stream Interface

In this section, we quantify the costs of a streaming interface and of enforcing delivery order. Figure 15 shows a breakdown of server-side CPU time when transferring tuples of varying sizes between two machines over a Tux connection. We compare unordered message delivery, ordered message delivery, and a stream-based interface. To eliminate IPC overhead, we ran Tux in pushdown mode; the callback simply copies data into an application buffer. Stream-based delivery is up to 1.96× slower than the message-based interface because the callback must parse tuples from a byte stream and copy data. Ordered and unordered message delivery have nearly identical costs in our runs: on AWS , packets arrive almost entirely in order for a flow (TCP/UDP), and our implementation detects this case and bypasses the reorder buffer (implemented with std::map in C++). Finally, the Tux-runtime cost increases with tuple size because larger tuples span more packets. Since Tux processes fixed-size packet batches per routine invocation, larger tuples trigger more invocations and context switches. We leave further runtime optimizations to future work.

To quantify the cost of ordering when packet reordering is needed, we simulate packet loss by dropping packets randomly on the sender side. The results (Figure 16) show significant latency advantages of unordered messaging over ordered delivery under packet loss conditions. At 1% packet loss rate, unordered messaging achieves 42.5%/39.2%/34.1% lower median/P90/P99 latency. At moderate 0.1% loss rates, unordered messaging reduces median/P99 latency by 11.1%/23.7%. This performance improvement stems from unordered messaging's ability to bypass head-of-line blocking and sorting that plague ordered protocols during packet loss events.

5.6 Impact of Preemption Frequency

We next study the impact of preemption on throughput and tail latency. We use a workload that consists of 99% Get and 1% scan operations on the Networked LeanStore server. The scan operation scans 5,000 records. We configure LeanStore to run request processing in message callback executing in the pushdown mode. The results are shown in Figure 18. We vary the preemption timeslice from 5μs to 1ms. Generally, small preemption timeslice results in lower throughput due to signal and context switch overhead. With a 5μs preemption timeslice, throughput is 40% lower compared to that of 500μs. After 100 μs timeslice, the throughput improvement plateaus as the preemption does not add significant overhead. On the other hand, with a 65μs timeslice, the p99 latency for Get operation is 2.5× lower compared to a 1ms timeslice. Smaller preemption timeslice also results in longer tail latency for scan, as short operations are prioritized. However, we find timeslice near 100μs is a sweet spot that balances throughput, tail latency for short-running callback function, and timely network processing.

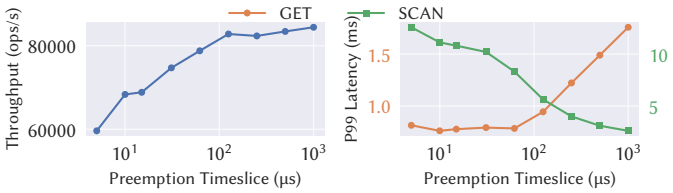


Figure 18: Impact of Preemption Timeslice on Throughput and Tail Latency: Small preemption timeslice generally results in better tail latency for short operations at the cost of impacting throughput and long operations. (We use a workload of 99% GET and 1% Scan operations with scan length of 5000.)

6 RELATED WORK

In this section, we cover related work not mentioned in Section 2.

DB-OS Co-design A significant body of research focuses on co-designing database systems to work more efficiently with the underlying OS. libdbos [65] and CumulusDB [46] explore running DBMS in privileged kernel space for designing more powerful abstractions. Another line of research focuses on optimizing scheduling within DBMS engine, including cooperative scheduling using coroutines to hide latency for memory and I/O [34, 36, 41, 49, 57] and task-based parallelism [52], as well as preemptive techniques like userspace interrupts to prioritize short transactions [37]. While these improve internal efficiency, Tux tackles the complementary bottleneck at the DB-OS network stack/scheduler boundary [66].

More Kernel Bypass Systems. IX [26], Arrakis [56] and Pegasus [55] redesign the OS dataplane by removing most kernel-mediated I/O or IPC via user-space data paths, runtimes, and specialized APIs. In contrast, Tux focus on stock Linux kernels and virtualized NIC hardware in the cloud rather than requiring new OS or kernel modules. MICA [48] holistically co-designs an in-memory KV store and its I/O path to maximize single-node throughput. In contrast, Tux is not an application redesign; it is a transport/runtime substrate that lets DBMSes gain similar benefits via message semantics and selective pushdown. Systems like eRPC [43] provide a high-level RPC interface, whereas Tux operates closer to the transport layer, offering both a compatible byte-stream interface and direct programmability via pushdown. Similarly, while Machnet [58] offers high-performance messaging, its sidecar architecture introduces IPC overhead that Tux's pushdown eliminates, and its in-order messaging can suffer from head-of-line blocking that Tux's unordered messaging avoids. Recent systems like Shinjuku [42], LibPreemptible [47], and Caladan [32] introduce microsecond-scale preemptive scheduling to improve tail latency and resource efficiency with the help of a specialized kernel module or user-space interrupt hardware. Tux can integrate or complement its preemptible runtime with these techniques to reduce preemption overhead.

7 CONCLUSION

We presented Tux, a high-performance drop-in networking stack for database systems. Tux achieves high performance without sacrificing compatibility by building on eBPF/XDP, enabling it to work with unmodified DBMSs while avoiding the complexity of kernel modules or exclusive NIC access. Through a message-based transport protocol and a pushdown abstraction for programmability, Tux eliminates key bottlenecks in the kernel network stack and OS scheduler. Our experiments show substantial improvements in throughput, latency, and CPU efficiency across real-world database systems, with minimal porting efforts. We believe Tux provides a practical and robust path forward for deploying high-performance, co-designed database networking in modern cloud.

References

[1] [n.d.]. Cassandra Stress. https://github.com/scylladb/cassandra-stress.

[2] [n.d.]. Chapter 53. Frontend/Backend Protocol - Pipelning. https://www.postgresql.org/docs/current/protocol-flow.html#PROTOCOL-FLOW-PIPELINING.

[3] [n.d.]. Cloud Cost Observability and Optimization. https://instances.vantage.sh/.

[4] [n.d.]. Coroutines. https://www.iso.org/standard/73008.html

[5] [n.d.]. Data Plane Development Kit. https://www.dpdk.org/

[6] [n.d.]. Data Plane Development Kit (DPDK) latency in Red Hat OpenShift - Part I. https://www.redhat.com/en/blog/dpdk-latency-red-hat-openshift-1

[7] [n.d.]. Deploying eBPF, XDP & AF_XDP for Cloud Native. https://archive.fosdem.org/2021/schedule/event/sdn_ebpf_afxdp/

[8] [n.d.]. eBPF and Network Trends Forecast for 2024. https://www.ebpf.top/en/post/network_and_bpf_2024/#11-exponential-growth-of-ebpf

[9] [n.d.]. F-Stack. https://github.com/F-Stack/f-stack

[10] [n.d.]. Goroutines. https://go.dev/doc/effective_go#goroutines

[11] [n.d.]. Kernel page-table isolation. https://en.wikipedia.org/wiki/Kernel_page-table_isolation

[12] [n.d.]. Max MTU of XDP programs. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_XDP/#__tabbed_1_1

[13] [n.d.]. MySQL X Protocol. https://dev.mysql.com/doc/dev/mysql-server/8.4.3/mysqlx_protocol_implementation.html.

[14] [n.d.]. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark

[15] [n.d.]. Placement groups for your Amazon EC2 instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html

[16] [n.d.]. Redis serialization protocol specification. https://redis.io/docs/latest/develop/reference/protocol-spec/.

[17] [n.d.]. A Remote Direct Memory Access Protocol Specification. https://datatracker.ietf.org/doc/html/rfc5040

[18] [n.d.]. ScyllaDB. https://docs.scylladb.com/manual/stable/kb/dpdk-hardware.html

[19] [n.d.]. ScyllaDB: No-Compromise Performance (Avi Kivity). https://www.youtube.com/watch?v=0S6i9BmuF8U&t=2586s

[20] [n.d.]. ScyllaDB Shard-per-Core Architecture. https://www.scylladb.com/product/technology/shard-per-core-architecture/

[21] [n.d.]. Setting up a xsk. https://docs.ebpf.io/linux/concepts/af_xdp/#setting-up-a-xsk

[22] [n.d.]. Stream Control Transmission Protocol. https://datatracker.ietf.org/doc/html/rfc4960

[23] [n.d.]. Stream Control Transmission Protocol. https://en.wikipedia.org/wiki/Stream_Control_Transmission_Protocol

[24] [n.d.]. Terabit Ethernet. https://en.wikipedia.org/wiki/Terabit_Ethernet.

[25] [n.d.]. VoltDB changes to adopt F-stack. https://github.com/DBOS-project/voltdb/tree/dpdk/src/frontend/org/voltcore/network

[26] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 49–65.

[27] Matthew Butrovich, Samuel Arch, Wan Shen Lim, William Zhang, Jignesh M. Patel, and Andrew Pavlo. 2025. BPF-DB: A Kernel-Embedded Transactional Database Management System For eBPF Applications. *To appear at SIGMOD 2025* (2025).

[28] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. 2023. Tigger: A database proxy that bounces with user-bypass. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3335–3348.

[29] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking {RocksDB}{Key-Value} Workloads at Facebook. In *FAST*. 209–223.

[30] Mark Cusack, John Adamson, Mark Brinicombe, Neil Carson, Thomas Kejser, Jim Peterson, Arvind Vasudev, Kurt Westerfeld, and Robert Wipfel. [n.d.]. Yellowbrick: An Elastic Data Warehouse on Kubernetes. ([n. d.]).

[31] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. 2024. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 55–73.

[32] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 281–297.

[33] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. {BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 487–501.

[34] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: coroutine-oriented main-memory database engine. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 431–444. https://doi.org/10.14778/3430915.3430932

[35] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 54–66.

[36] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The art of latency hiding in modern database engines. *Proceedings of the VLDB Endowment* 17, 3 (2023), 577–590.

[37] Kaisong Huang, Jiatang Zhou, Zhuoyue Zhao, Dong Xie, and Tianzheng Wang. 2025. Low-Latency Transaction Scheduling via Userspace Interrupts. (2025).

[38] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 466–481.

[39] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, and Carsten Binnig. 2025. A Wake-Up Call for Kernel-Bypass on Modern Hardware. In *Proceedings of the 21st International Workshop on Data Management on New Hardware*. 1–5.

[40] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. {mTCP}: a highly scalable user-level {TCP} stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 489–502.

[41] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the" killer nanoseconds". *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714.

[42] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for {μsecond-scale} Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.

[43] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter {RPCs} can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16.

[44] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[45] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4536–4545.

[46] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *PVLDB* 17 (2024).

[47] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. 2024. Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 922–936.

[48] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. {MICA}: A holistic approach to fast {In-Memory}{Key-Value} storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 429–444.

[49] Haotian Liu, Runzhong Li, Ziyang Zhang, and Bo Tang. 2024. Tao: Improving Resource Utilization while Guaranteeing SLO in Multi-tenant Relational Database-as-a-Service. *Proceedings of the ACM on Management of Data* 2, 4 (2024), 1–26.

[50] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. 2019. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 399–413.

[51] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 221–235.

[52] Jan Mühlig and Jens Teubner. 2021. MxTasks: How to Make Efficient Synchronization and Prefetching Easy. In *Proceedings of the 2021 International Conference on Management of Data*. 1331–1344.

[53] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[54] John Ousterhout. 2021. A linux kernel implementation of the homa transport protocol. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 99–115.

[55] Dinglan Peng, Congyu Liu, Tapti Palit, Anjo Vahldiek-Oberwagner, Mona Vij, and Pedro Fonseca. 2025. Pegasus: Transparent and Unified Kernel-Bypass Networking for Fast Local and Remote Communication. In *Proceedings of the Twentieth European Conference on Computer Systems*. 360–378.

[56] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System Is the Control Plane. 33, 4 (2015). https://doi.org/10.1145/2812806

[57] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with coroutines: a practical approach for robust index joins.

Proceedings of the VLDB Endowment 11, 2 (2017), 230–242.

[58] Alireza Sanaee, Vahab Jabrayilov, Ilias Marinos, Anuj Kalia, Divyanshu Saxena, Prateesh Goyal, Kostis Kaffes, and Gianni Antichi. 2025. Fast Userspace Networking for the Rest of Us. *arXiv preprint arXiv:2502.09281* (2025).

[59] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. 2020. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE micro* 40, 6 (2020), 67–73.

[60] V Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. 2016. Aerospike: Architecture of a real-time operational dbms. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1389–1400.

[61] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[62] Nishant Suneja. 2019. Scylladb optimizes database architecture to maximize hardware performance. *IEEE Software* 36, 04 (2019), 96–100.

[63] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.

[64] 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. 2021. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 195–211.

[65] Xinjing Zhou, Viktor Leis, Jinming Hu, Xiangyao Yu, and Michael Stonebraker. 2025. Practical db-os co-design with privileged kernel bypass. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.

[66] Xinjing Zhou, Viktor Leis, Xiangyao Yu, and Michael Stonebraker. 2025. OLTP Through the Looking Glass 16 Years Later: Communication is the New Bottleneck. In *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, Netherlands*.

[67] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. 2024. {DINT}: Fast {In-Kernel} Distributed Transactions with {eBPF}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 401–417.

[68] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, et al. 2023. Deploying user-space {TCP} at cloud scale with {LUNA}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 673–687.

[69] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. 2022. Efa: A viable alternative to rdma over infiniband for dbmss?. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 1–5.