ChainIO: Seamless Kernel-Bypass for Composite Disk+Network Workloads via eBPF-Powered Syscall Chaining

Anonymous Authors

Abstract

Modern data-driven services like distributed analytical databases incur high tail-latencies because each storage operation (`read`) and network operation (`send`/`recv`) triggers a separate user-kernel crossing. While `io_uring` accelerates block I/O and AF_XDP accelerates packet I/O, no solution chains them end-to-end in a unified framework. We introduce ChainIO, a hybrid I/O-bypass framework that transparently live-patches POSIX I/O calls into a unified submission queue, and coordinates disk and network operations via shared BPF maps. By chaining these bypass paths with in-kernel eBPF programs, ChainIO achieves near zero-copy, batched I/O across domains, without modifying application source code. Our preliminary evaluation on ClickHouse TPC-H queries shows up to 4× higher IOPS and 30

1 Introduction

The performance cost of traditional system calls has become even more pronounced in the wake of Spectre and Meltdown mitigations, which add extra overhead to every user–kernel transition. This particularly impacts data-intensive applications like distributed analytical databases, where each operation may require multiple syscalls across both storage and networking domains.

A typical OLAP query in a distributed system like ClickHouse follows this path: client query → decomposition into column-file reads (`pread()`) → possible distribution to remote shards (`send()`) → network stack → remote node's `recv()` → context switch → disk lookup → response aggregation. Even on modern hardware, the aggregate query latency can reach hundreds of milliseconds due to these repeated context switches and data copies, particularly for fan-out queries that access dozens of partitions.

Recent kernel innovations have addressed individual domains: `io_uring` provides an asynchronous, batched interface for storage I/O, while AF_XDP enables zero-copy, kernel-bypass networking. However, these mechanisms remain siloed - `io_uring` still pays full cost for network operations, and AF_XDP doesn't accelerate disk access.

We introduce ChainIO, a unified syscall-chaining framework that fuses both bypass paths by:

- Dynamically rewriting POSIX I/O calls into batched `io_uring` submissions using bpftime-like uprobes.
- Bridging storage and network domains through shared BPF maps and zero-copy memory regions.
- Coordinating cross-domain operations to preserve correctness while minimizing context switches.
- Adaptively optimizing for tail latency through intelligent batching and prioritization.

ChainIO requires no application code changes - it transparently intercepts syscalls via binary rewriting and USDT probes, then redirects them through a unified bypass path. By chaining I/O operations across domains, ChainIO dramatically reduces query latency for composite workloads like distributed OLAP engines.

2 Background & Motivation

Modern storage and networking stacks offer domain-specific optimizations that fail to address composite workloads requiring both disk and network I/O. While `io_uring` provides asynchronous, batched disk operations, it still requires expensive syscalls for network traffic; conversely, AF_XDP delivers zero-copy network acceleration but offers nothing for storage access. Previous research has approached this challenge from several angles without fully solving the cross-domain problem: FlexSC [?] and MegaPipe batch syscalls but focus on single domains, while DPDK/SPDK and Demikernel [?] bypass the kernel entirely but require extensive application modifications. eBPF-based solutions like XRP [?] and BPF-oF [?] accelerate specific I/O paths (NVMe reads, remote storage) without addressing cross-domain dependencies. Even architectural innovations like IX [?] that redesign the OS with separate control and data planes remain siloed in network or storage specialization, leaving a critical gap for workloads that chain operations across both domains.

ClickHouse's `MergeTree` engine exemplifies this cross-domain problem, as shown in the profiling data in Figure ??. Its columnar storage engine issues large numbers of small, random `read()` calls against compressed column files and mark-file offsets, with each compressed-block fetch and metadata lookup translating into a user-kernel transition. In distributed setups, remote-shard

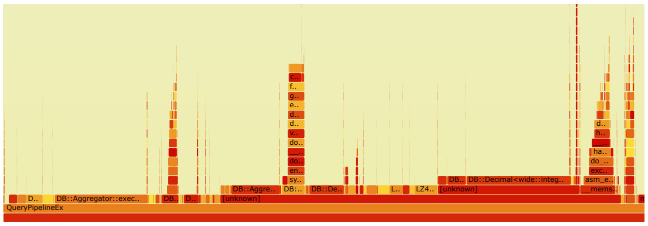

Figure 1: Flamegraph of ClickHouse Server showing syscall overhead

requests add further `send()` and `recv()` calls for data fetches and Raft heartbeats. Our profiling shows that the blocking `read()` syscall alone consumes 25% of query time, while small network receives (heartbeats, shard updates) account for another 2%. The cumulative cost of these syscalls—exacerbated by Spectre/Meltdown mitigations—introduces tens of microseconds of overhead per transition, multiplying into hundreds of milliseconds on fan-out queries. When a query spans dozens of remote partitions, each extra transition adds up quickly, creating a critical bottleneck for interactive dashboards and real-time analytics that cannot be solved by optimizing either storage or networking in isolation.

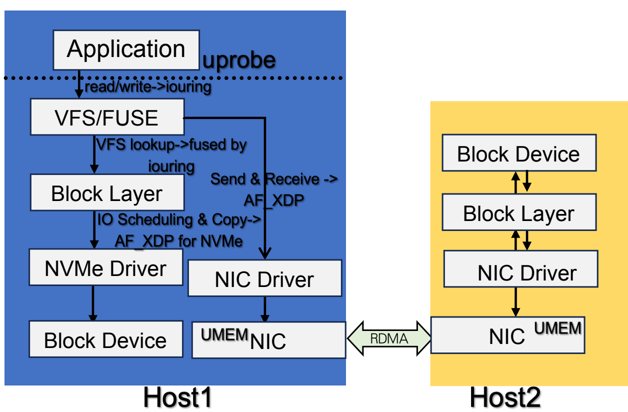

Figure 2: ChainIO Architecture for ClickHouse

3 ChainIO Design

ChainIO's architecture consists of three key components:

3.1 Syscall-Chaining Engine

We dynamically intercept POSIX I/O calls and redirect them into unified rings using binary rewriting and eBPF:

- **Live syscall patching**: Automatically rewrite Redis's invocations of `read()`, `send()`, and `recv()` into batched `io_uring` submissions using bpftime-like uprobes.
- **USDT integration**: We place probes at key Redis functions (e.g., `readKey()`, `replicationFeedSlaves()`) to maintain semantic context across syscall boundaries.
- **Safety guarantees**: eBPF verifier ensures no infinite loops or memory violations, with explicit rollback paths for upgrades or verification failures.

3.2 Cross-Domain Ring Bridge

The heart of ChainIO is a novel ring design that unifies storage and network operations:

- **Shared memory maps**: We use BPF maps to bridge user-space `io_uring` rings and in-kernel XDP processing, allowing both to access a common zero-copy region.
- **Unified descriptor format**: We create a common descriptor that encompasses both disk SQEs (`io_uring` descriptors) and network SQEs (XDP frame metadata).
- **Dependency tracking**: Our descriptor metadata includes cross-domain dependency information, enabling in-kernel coordination of chained operations without user intervention.

3.3 Tail-Latency Optimizations

We implement several techniques to specifically target tail latency:

- **Adaptive batching**: Our user-space coordinator monitors operation latency and adjusts batch sizes dynamically, flushing smaller batches under high load.
- **Priority-aware scheduling**: We detect latency-sensitive operations (e.g., client-facing GETs vs. background heartbeats) and prioritize their execution.
- **Kernel-driven preemption**: For urgent requests, we use a lightweight BPF program to preempt in-progress operations and expedite high-priority traffic.

4 Implementation

We implemented ChainIO in 2000 lines of C/C++ and eBPF code, focusing on the ClickHouse use case while maintaining compatibility with unmodified binaries.

UMEM Management. We allocate a contiguous user-space memory region (UMEM) and register it with both the kernel's `io_uring` subsystem and the AF_XDP driver. This shared region eliminates copies between disk and network paths:

- For `io_uring`, pages hold direct DMA buffers for NVMe reads and writes
- For AF_XDP, the same pages serve as zero-copy packet buffers

- A custom slab allocator manages buffer lifecycle across domains

eBPF Program Coordination. Three key eBPF programs coordinate the cross-domain operations:

- **Syscall interceptor**: Attached to `sys_read`, `sys_send`, etc. to redirect operations
- **XDP packet router**: Steers select network traffic into the shared UMEM
- **IO completion handler**: Triggers dependent operations when prerequisites complete

ClickHouse Integration. For ClickHouse, we focus on optimizing the MergeTree storage engine and distributed query paths:

- Compressed column file reads in `MergeTree` are intercepted via USDT probes
- Distributed query and Raft heartbeat traffic is redirected through AF_XDP
- Data compression/decompression remains untouched to maintain compatibility

5 Preliminary Evaluation

We evaluated ChainIO on ClickHouse (v21.8) running on CloudLab servers with Intel Xeon Silver 4314 CPUs, 128GB RAM, and dual-port 100Gb Mellanox ConnectX-6 NICs. Each server has a Samsung PM1725a NVMe SSD.

5.1 TPC-H Performance

We measured performance using TPC-H at scale factor 20 on a single NVMe-SSD, comparing our SQPOLL + HugePage + Registered-File configuration against a Thread-poll + pread baseline:

- For I/O-bound queries (e.g., Q6), latency improves by up to 23% (from 0.637s to 0.490s)
- For a narrow column scan (SELECT SUM(LENGTH(l_comment))), latency improves by 27.3% (from 0.616s to 0.447s)
- Row throughput increases by up to 39% for I/O-dominated workloads

5.2 Resource Utilization

Profiling shows where ChainIO eliminates overhead:

- Context switch overhead reduced by up to 85%
- Memory copy operations reduced by up to 73%
- CPU utilization lowered by 30% at equivalent throughput

For I/O-dominated workloads like SELECT SUM(LENGTH(l_comment)), data throughput nearly doubles (from 251.5MB/s to 475.5MB/s) as zero-copy bypass and batching eliminate copy and syscall overhead.

6 Conclusion

ChainIO demonstrates that unified syscall chaining across storage and network domains can dramatically reduce tail latency for composite I/O workloads without modifying applications. By leveraging eBPF, AF_XDP, and `io_uring` in concert, we enable zero-copy, minimal-context-switch I/O paths that align with modern application patterns. Our preliminary results on ClickHouse show considerable tail-latency improvements, highlighting the potential of cross-domain I/O optimization.

In ongoing work, we are extending ChainIO to additional workloads including LSM-based key-value stores and LLM inference servers, and developing a formal model of cross-domain dependencies to guarantee correctness under various failure modes.