Tigger: A Database Proxy That Bounces With User-Bypass

Matthew Butrovich
mbutrovi@cs.cmu.edu
Carnegie Mellon University

Karthik Ramanathan
Carnegie Mellon University

John Rollinson
john.rollinson@westpoint.edu
Army Cyber Institute

Wan Shen Lim
wanshenl@cs.cmu.edu
Carnegie Mellon University

William Zhang
wz2@cs.cmu.edu
Carnegie Mellon University

Justine Sherry
sherry@cs.cmu.edu
Carnegie Mellon University

Andrew Pavlo
pavlo@cs.cmu.edu
Carnegie Mellon University

ABSTRACT

Developers often deploy database-specific network proxies whereby applications connect transparently to the proxy instead of directly connecting to the database management system (DBMS). This indirection improves system performance through connection pooling, load balancing, and other DBMS-specific optimizations. Instead of simply forwarding packets, these proxies implement DBMS protocol logic (i.e., at the application layer) to achieve this behavior. Consequently, existing proxies are user-space applications that process requests as they arrive on network sockets and forward them to the appropriate destinations. This approach incurs inefficiencies as the kernel repeatedly copies buffers between user-space and kernel-space, and the associated system calls add CPU overhead.

This paper presents user-bypass, a technique to eliminate these overheads by leveraging modern operating system features that support custom code execution. User-bypass pushes application logic into kernel-space via Linux's eBPF infrastructure. To demonstrate its benefits, we implemented Tigger, a PostgreSQL-compatible DBMS proxy using user-bypass to eliminate the overheads of traditional proxy design. We compare Tigger's performance against other state-of-the-art proxies widely used in real-world deployments. Our experiments show that Tigger outperforms other proxies — in one scenario achieving both the lowest transaction latencies (up to 29% reduction) and lowest CPU utilization (up to 42% reduction). The results show that user-bypass implementations like Tigger are well-suited to DBMS proxies' unique requirements.

PVLDB Reference Format:
Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. Tigger: A Database Proxy That Bounces With User-Bypass. PVLDB, 16(11): 3335 - 3348, 2023.
doi:10.14778/3611479.3611530

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

1 INTRODUCTION

Modern cloud applications often connect to database management systems (DBMSs) over a network in a manner that is not optimal for performance: either maintaining large numbers of persistent, mostly idle connections or rapidly churning connections. Both scenarios cause the DBMS to squander resources on state information for each connection or perform onerous and redundant network operations. One standard solution to overcome this problem is to use a *proxy* that acts as a middlebox [37] between the front-end client and back-end DBMS. Examples include AWS RDS Proxy [2], PgBouncer [19], Pgpool-II [20], MaxScale [11], and ProxySQL [23]. Although their features vary, these proxies all implement a target DBMS's network protocol so that clients can connect to them without needing to modify application code.

Using a DBMS proxy as the authoritative connection manager for the back-end DBMS provides multiple benefits. The proxy maintains a persistent connection pool to the back-end DBMS to multiplex client requests and responses — minimizing the number of connections talking to the DBMS. This optimization reduces the overhead of tasks that scale linearly with the number of clients like transaction commit. Furthermore, because the proxy's pooled connections only authenticate with the DBMS once, it prevents the DBMS from repeatedly performing connection setup procedures and frees system resources for query execution.

But the need to support DBMS network protocols imposes constraints on proxies' implementations. As Application Layer (L7) [42] middleboxes, these proxies follow the same design: they are user-space applications that read byte streams from client sockets, extract connection state, match the client to a back-end socket, and forward messages to the correct destination. This design incurs significant overhead from the repeated operating system (OS) system calls to read and write to sockets. Some high-performance systems have previously used kernel-bypass to elide the OS networking stack [24]. Unfortunately, this approach increases engineering and deployment complexity. To our knowledge, no DBMS proxies employ this technique; they are inefficient user-space applications that do not scale effectively for the most demanding applications.

Given this, we present an alternative system architecture called **user-bypass**. User-bypass relies on eBPF programs that push down DBMS-specific connection logic into the OS to create fast paths in the Linux networking stack. User-bypass elides expensive system calls and buffer copying without sacrificing functionality or safety. We are not the first to embed L7 logic into the kernel using eBPF [51, 60, 72]. However, to our knowledge, we are the first to articulate the idea of "user-bypass" instead of "kernel-bypass" and the first to deploy this design strategy to DBMS proxies. To demonstrate the effectiveness of this approach, we implemented a PostgreSQL-compatible proxy called **Tigger** using user-bypass. It offers efficient connection pooling and workload mirroring within kernel-space.

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611530

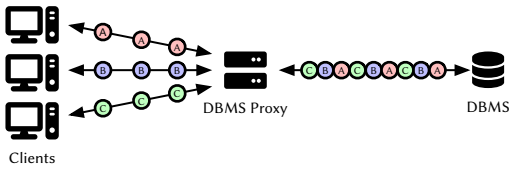


Figure 1: Connection Pooling Example – A DBMS proxy reduces the number of connections to back-end DBMSs by multiplexing connections from many clients to a smaller number of pooled connections.

Tigger supports the target DBMS's network protocol and is a drop-in replacement for other proxies.

We compare Tigger's performance against other open-source proxies for online transaction processing (OLTP) workloads. Our results show that Tigger's user-bypass architecture is the most responsive and reduces transaction latency by up to 29%. Tigger also minimizes CPU utilization — offering the best performance by being 42% more efficient than the next proxy. When comparing Tigger's workload mirroring capabilities, it provides an average of 92% lower transaction latency while using 88% less CPU.

Our work makes the following contributions: (1) a discussion of DBMS proxies in the context of modern cloud applications, (2) the user-bypass method for pushing DBMS logic into kernel-space, and (3) the design of the Tigger DBMS proxy using user-bypass.

2 BACKGROUND

We begin with motivating the need for DBMS proxies. We present the challenges of scaling the number of persistent DBMS clients and their overhead, and then describe how proxies use pooling to improve DBMS efficiency. Lastly, we discuss existing proxies implementations and detail their scalability limits in cloud environments.

2.1 Connection Scaling

Modern cloud application frameworks perform horizontal scaling in response to changes in load, creating more instances that must communicate with the DBMS [3, 16]. Thus, a short burst of activity can quickly generate thousands of connections. Some programming frameworks and libraries use client-side pooling to maintain long-lived connections (e.g., Phoenix [63], HikariCP [6], pgxpool [21]). This scenario amplifies the scale of connections to the DBMS, as new instances in an autoscaling group can result in thousands of connections for their pools that may sit idle.

For multiple reasons, supporting many persistent clients is challenging for DBMSs. The first is that each new connection incurs some fixed overhead in the DBMS even before that connection issues a query request. To support multiple connections in parallel, the DBMS will assign each connection a *worker* that is either a lightweight thread (e.g., MySQL, Oracle, MSSQL) or a heavyweight process (e.g., PostgreSQL, DB2, TimescaleDB). Each worker requires a fixed amount of memory, CPU, and effort from the OS to manage those resources. For example, PostgreSQL has a memory overhead on the order of megabytes per connection [49, 66]. This cost is for a new, unused connection before the DBMS populates auxiliary data structures, such as result caches or prepared statement digests.

Second, since a DBMS provides ACID guarantees for transactions, its concurrency control scheme requires it to perform additional

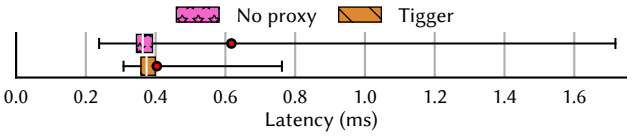


Figure 2: Connection Pooling for Many Clients – YCSB transaction latencies showing the effect of Tigger's transaction pooling. The red circle shows sample mean, and the upper whisker shows p99.

work for each connected client. For example, with optimistic concurrency control, the DBMS performs a validation step when a transaction commits to check whether that transaction conflicts with the read/write sets of other clients connected to the DBMS [85]. PostgreSQL allocates memory for each possible connection, so the conventional wisdom is to set its `max_connections` knob as small as possible to keep concurrency control operations fast; otherwise, it limits the scalability of the DBMS [48].

One approach to handle many connections is to scale the DBMS horizontally by adding more nodes. Then, an L3/L4 proxy (e.g., HAProxy [10], nginx [18]) performs load balancing over TCP sessions between nodes. However, these additional nodes add complexity and cost to a deployment. Combined with client-side pooling, the DBMS could waste resources supporting mostly idle connections. A better approach to address these scaling challenges is to deploy a DBMS proxy that performs *connection pooling* [61]. Instead of connecting to the DBMS, applications connect to the proxy using the same authentication methods. The proxy then manages all connections between front-end clients and one or more back-end DBMSs. With *transaction pooling*, the proxy multiplexes client connections over shared back-end connections, as shown in Figure 1. The proxy temporarily allocates a server connection until a transaction commits or rolls back. This approach improves performance when many persistent connections do not continuously submit transactions to the DBMS.

To demonstrate how transaction pooling can improve performance, we run the YCSB workload [40] with 10,000 connections in two configurations. The clients connect directly to the DBMS in the first setup and through Tigger DBMS proxy in the second. We discuss our experimental setup in Section 6. Figure 2 shows the latency distributions from the two scenarios. YCSB's p99 and mean latencies are much higher than the median when many clients connect directly to the DBMS. In contrast, using the proxy slightly increases the minimum latency while significantly reducing the p99 and mean latencies. We explore more workloads in this configuration in Section 6.2.

2.2 Connection Establishment

In addition to large numbers of persistent connections, short-lived ephemeral clients also pose challenges for DBMSs. Ideally, applications should use client-side connection pooling, but improperly tuned web frameworks result in short client *sessions* (i.e., the time between connect and disconnect) that last only for the duration of a transaction. Popular frameworks, such as Laravel [27] and Django [28], do not use persistent connections by default. The application opens a new connection to the DBMS for each request and discards the connection upon completion. Furthermore, some cloud software designs cannot exploit client-side connection pooling. For

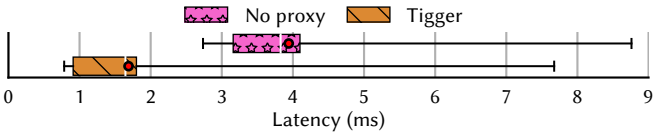


Figure 3: Connection Pooling for Serverless Clients – YCSB transaction latencies showing the effect of Tigger's persistent connections. The red circle shows sample mean, and the upper whisker shows p99.

example, microservices that rely on stateless serverless functions (e.g., AWS Lambda) create a new connection on each invocation.

Short sessions are problematic in DBMSs because the connection setup and teardown work steal CPU cycles that the DBMS could use for executing queries. This overhead is due to how DBMSs handle parallel connections: the system creates a new worker dedicated to each connection. Creating a new worker incurs overhead for (1) task creation, (2) socket allocation, (3) TCP handshakes, (4) Transport Layer Security (TLS) handshakes, (5) client authentication, and (6) querying DBMS settings and catalogs. These unavoidable steps take milliseconds and waste DBMS resources when performed thousands of times per second. Then when the connection closes, the DBMS performs additional bookkeeping and cleanup.

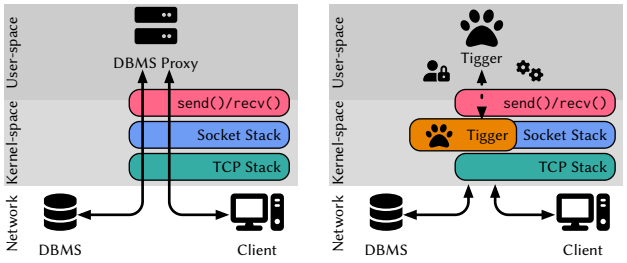
Once again, DBMS proxies help with this problem. Since they are optimized for connection management, their logic for opening and closing connections is more efficient than DBMS workers that have to balance many tasks (e.g., query planning/execution, logging, garbage collection). Moving connection setup and teardown execution to a different machine frees the DBMS's CPU to focus on these other tasks. The extra network hop introduced by the DBMS proxy offsets costly system calls like `fork()` and the subsequent page faults accompanying a new process.

To highlight these issues, we use PostgreSQL and YCSB in two configurations representative of serverless applications where each transaction opens a new connection. In the first setup they connect directly to the DBMS, and in the second they connect through the Tigger DBMS proxy. Figure 3 shows the latency improvement at 2,000 TPS when serverless clients connect to Tigger instead of directly to the DBMS. PostgreSQL uses processes for multitasking, so every transaction performs the `fork()` and `wait()` system calls. As a baseline not shown in the figure, the mean latency for YCSB with persistent servers in this environment is 0.2 ms. Creating and discarding a connection in PostgreSQL adds over 3.5 ms of latency.

DBMS proxies can provide partial benefits of a persistent connection pool even when it is not an option. Despite introducing extra network hops, Tigger reduces the latency overhead of short-lived connections — dropping from an average of 3.9 ms to 1.7 ms. For short transactions like in YCSB, a DBMS proxy halves the latency in serverless environments. These results demonstrate how connection establishment affects performance.

2.3 User-Space Proxy Design

DBMS proxies contain logic specific to a back-end DBMS's network protocol, thus making them L7 network applications. This behavior contrasts with L3/L4 proxies (e.g., HAProxy [10], nginx [18]) that transparently perform load balancing at lower layer protocols. Such L7 logic requires processing network data in user-space (i.e., above the OS networking stack) or applying deep packet inspection (DPI) in lower levels of the network stack.



(a) User-space proxy **(b)** User-bypass proxy

Figure 4: User-Space and User-Bypass DBMS Proxies – User-space proxies rely on system calls to redirect queries and results between clients and DBMSs. Tigger employs user-bypass as a fast path in kernel-space, only passing authentication and user settings message to user-space.

To the best of our knowledge, all existing DBMS proxies follow the same design in Figure 4a: they are event-driven user-space applications that, after the authentication steps, (1) read client messages from a network socket, (2) inspect the stream of bytes, (3) match the client to a back-end server, and (4) send the data on the matched socket. Query results follow a similar logic but with the sender and receiver reversed. Our inspection of open-source proxies supports this belief, though implementation details vary from proxy to proxy. PgBouncer is written in C and uses the libevent library for notifications but is entirely single-threaded. Odyssey [14] is also in C, directly calls `epoll` for event management, but uses a bespoke coroutine library written in assembly and asynchronous IO to enable parallelism. ProxySQL is in C++, relies on the `poll` system call, and follows a more typical multi-threaded design.

These proxy implementations coordinate `send()` and `recv()` system calls with DBMS protocol-specific control logic. This design limits their scalability as network bandwidths increase. Research shows that copying buffers during system calls accounts for ~50% of the kernel's network stack CPU cycles [36]. PostgreSQL developers concluded that a saturated PgBouncer process spends most of its time copying data in and out of buffers between user-space and kernel-space [38]. As we will show in Section 6, PgBouncer's single-threaded design limits its throughput. Although Odyssey offers more parallelism, its CPU demands scale with its capabilities. Ideally, a DBMS proxy should do most of its work without copying socket buffers to user-space via system calls.

3 USER-BYPASS

We next define our user-bypass method in the context of Linux networking and existing kernel-bypass techniques. We also discuss the kernel feature that enables user-bypass and its limitations.

3.1 Kernel-Bypass vs. User-Bypass

A complete overview of Linux's networking stack is beyond the scope of this paper; we will focus on the portion relevant to DBMS proxies. Linux contains multiple layers for processing network traffic. These layers handle transport (e.g., TCP), network (e.g., IP), and link layer (e.g., Ethernet) protocols. The stack exposes a socket interface [25] for applications to copy data between user-space and

kernel-space, along with traffic control [26] interfaces to configure queuing disciplines and network filters.

User-space applications that prioritize performance over simplicity can elide these software layers using *kernel-bypass* methods. In this scenario, a user-space application receives bytes directly from the device driver — bypassing the kernel's network stack. Thus, the application manages communication protocols and their associated state machines. The most common kernel-bypass pattern for network applications is to use Intel's Data Plane Development Kit (DPDK) [17] software library with user-space TCP implementations like mTCP [56]. Other libraries (e.g., F-Stack [5] and ScyllaDB's Seastar [24]) attempt to simplify development by bundling DPDK with bespoke TCP logic. In 2018, Linux added native kernel-bypass support with AF_XDP [54], removing the dependency on out-of-tree kernel modules such as DPDK. We discuss our (frustrating) experiences using kernel-bypass in DBMS proxies in Section 8.

Historically, kernel-bypass was the preferred way to implement high-performance Linux networking applications. There are several reasons for this view: (1) the Linux networking stack was perceived as slow and inefficient, (2) applications performed encryption in user-space using a software library (e.g., GnuTLS, LibreSSL, OpenSSL), and (3) a lack of programmability in the networking stack. The interfaces to program the network stack were limited to filtering and routing decisions based on L2/L3 rules (e.g., `iptables`, `nftables`, `tc`). This lack of extensibility in the kernel prevented L7 DPI necessary for a DBMS proxy. Thus, kernel-bypass remained viable for network applications with complex user-space logic.

Kernel-bypass aims to improve performance by transferring buffers between devices and user-space applications instead of the kernel processing them. *User-bypass* is the opposite approach: the developer pushes application logic into the kernel's network stack as low as possible to avoid copying data between user-space and kernel-space, while benefiting from the kernel handling L1-L4 networking. Until recently, if a developer wanted to embed application logic into the kernel, they would have to either (1) load a kernel module or (2) modify and recompile it. Such approaches are difficult and sacrifice system reliability and safety. However, updates to Linux make user-bypass a viable alternative to kernel-bypass. The efficiency of the Linux networking stack has also improved: a single CPU core can process 42 Gbps [36], and dedicated servers can process 670 Gbps of data [29]. Next, kernel TLS (kTLS) allows developers to move encryption into kernel-space and hardware [71]. But the key reason that user-bypass is now possible is the increased programmability via eBPF, which is composable with kTLS [33].

3.2 eBPF

To understand user-bypass, we now provide an overview of how the technique embeds DBMS logic in the Linux kernel. The primary technology that enables this functionality is *extended Berkeley Packet Filter* (eBPF). This modern Linux subsystem enables developers to write safe, event-driven programs running in kernel-space [76]. Application developers and cloud vendors have rapidly adopted eBPF due to its safety and features [7, 9, 12]. For example, Meta loads over 40 eBPF programs on every server, with hundreds more loaded on demand [77].

eBPF programs run a limited instruction set in a kernel-embedded virtual machine. Developers typically write eBPF programs in higher-level languages like C or Rust that compile to eBPF bytecode via LLVM. Upon loading the eBPF program, modern kernels compile the bytecode to native machine code. eBPF program capabilities vary based on their type and attachment point, but they attach to predetermined functions in the OS stack for network processing.

Developers load eBPF programs into kernel-space and then associate them with events (e.g., functions, static tracepoints) to trigger their execution. When a running thread hits the attachment point, it starts the execution of the eBPF program in privileged mode. Depending on their behavior and attachment location, developers use eBPF programs for software debugging, profiling, or modifying data flow (e.g., network buffers) through kernel-space. Tigger attaches eBPF programs with DBMS protocol logic in the socket and Traffic Control (TC) layers of the Linux networking stack.

The execution state of eBPF programs is ephemeral, meaning that its decision-making is limited to the data available during a single invocation of the handler. However, with *eBPF maps*, the program can maintain state across events, enabling user-bypass to support more complex application behavior.

These data structures reside in kernel-space and are the primary mechanism for creating stateful user-bypass programs. Tigger relies on three eBPF map types to coordinate execution across multiple eBPF programs: the (1) stack and (2) array map types are a persistent stack and array, respectively, while the (3) sockmap type is a unique map type that attaches an eBPF program to socket activity. Developers associate a sockmap with a single eBPF program and then add or remove socket file descriptors to or from the map. When activity occurs on a registered socket (i.e., updates to an ingress or egress buffer), the kernel executes the associated eBPF program.

Because the CPU is in privileged mode when eBPF programs run, the kernel requires them to pass a verification step before it loads them. The eBPF verifier enforces kernel API compliance, memory access safety, execution bounds, and instruction count. The verifier generates a control flow graph for all possible branches of the eBPF program and checks limits like 512 B stack size and 1m instructions. Although these restrictions limit the complexity of application logic with user-bypass, DBMS network protocols for the most common message types (i.e., queries, results) are expressible in eBPF. Despite the verifier's guarantees, developers should consider security and safety practices with eBPF deployment due to its close interaction with kernel functions, especially in multi-tenant environments.

4 TIGGER DBMS PROXY

We present Tigger as a solution that relies on user-bypass to overcome the challenges with existing DBMS proxies described in Section 2.3. As shown in Figure 4b, Tigger employs user-bypass, resulting in both user-space and kernel-space components to implement the DBMS's network protocol.

In this section, we first show how to apply user-bypass to DBMS proxies with our implementation for Tigger. We then detail Tigger's support for the PostgreSQL network protocol.

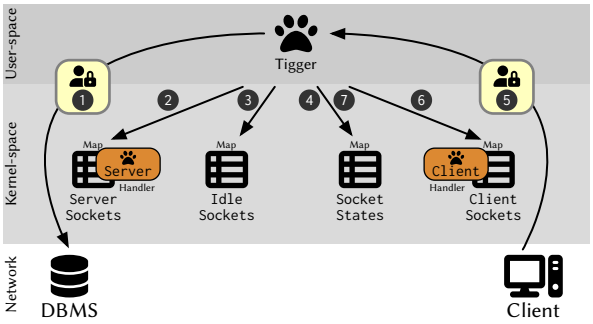


Figure 5: Tigger's User-Bypass Architecture – Tigger's hybrid design constrains user-space and kernel-space (i.e., eBPF programs and maps) components. We describe the steps in Section 4.1

.

4.1 User-Bypass Proxy Design

Tigger is a modified version of PgBouncer [19] that employs user-bypass to replace core components with eBPF-enhanced implementations. We chose PgBouncer as the foundation for Tigger because it is the most widely deployed DBMS proxy for PostgreSQL. However, the user-bypass design in this section could be applied to a different proxy (e.g., MaxScale for MySQL) to support other DBMSs. User-bypass results in a hybrid software design with both user-space and kernel-space logic, providing a fast path for the most common proxy tasks. Tigger's user-space component retains PgBouncer's single-threaded design but achieves parallelism through user-bypass because eBPF components run on kernel threads.

The user-space portion of Tigger is responsible for connection establishment, client authentication, and settings management. This component synchronizes the connection state with Tigger's kernel-space logic by reading and writing to eBPF maps. Tigger retains these parts as the user-space component for several reasons: (1) DBMS authentication methods (e.g., SCRAM-SHA-256, GSSAPI) are too complex to satisfy the eBPF verifier's limitations described in Section 3.2, (2) these events are infrequently executed and not part of the hot path of dispatching queries, and (3) it simplifies user-bypass engineering by reusing an existing application. These user-space authentication components follow the same semantics as in PgBouncer, enabling administrators to define how front-end users map to back-end connection pools. These settings ensure clients cannot submit queries to the DBMS with improper credentials.

Tigger maintains two independent connection pools: (1) a user-bypass pool (2) a user-space pool. The first is dedicated to user-bypass links between clients and back-ends. In the typical case, Tigger links a client to a back-end DBMS, redirects the session's queries and responses, and unlinks the two endpoints without executing any user-space code. In the rare event that all user-bypass sockets are in use, Tigger can pass a client's session up to the fallback connection pool managed in user-space at the cost of being slower. Exceptional protocol operations like cancel requests also pass to user-space and route through this pool.

When Tigger starts, it loads its eBPF maps and programs into kernel-space. Tigger's connection pooling relies on two eBPF *handler* programs: one to process buffers for back-end DBMS sockets (`Server`) and another for front-end client sockets (`Client`). Each handler attaches to its sockmap to trigger execution, and the other

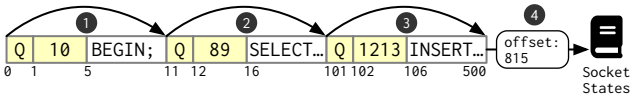


Figure 6: Applying DBMS Protocol Logic – Tigger performs DPI on DBMS messages to determine types and lengths. If a message spans multiple buffers, Tigger stores the position to start reading the next buffer in the `SocketStatesMap` eBPF map. We describe the steps in Section 4.2.

two maps (`IdleSocketsMap` and `SocketStatesMap`) synchronize the kernel-space state with the user-space components. Depending on the proxy's configuration, Tigger may install more eBPF handlers to provide additional features (see Section 5.2).

Most of Tigger's handlers operate at the socket layer (i.e., above the TCP stack) attached to sockmap events. The handlers run on kernel worker threads responsible for software interrupts, which is how Linux processes network events in kernel-space. Although eBPF programs can hook into lower levels of the networking stack (e.g., XDP, TC) it is not ideal to push DBMS protocol logic lower than the socket layer: every layer bypassed in kernel must be re-implemented by the proxy. For example, hooking into the network stack below the TCP layer requires the proxy to implement the complex TCP state machine and its associated messages (e.g., ACKs and retries). Furthermore, eBPF programs must be attached at the socket layer to perform DPI on encrypted content using kTLS [33].

At the sockmap layer, Tigger benefits from the OS handling protocols below the L7 layer. The OS arranges ingress bytes in their correct sequence order and reliably sends egress bytes. Socket buffers appear as they would in user-space (i.e., ready for L7 logic), with the OS processing headers related to TCP/IP and Ethernet. Therefore, Tigger's eBPF handlers' logic is similar to user-space DBMS proxy logic. After compilation, Tigger's `Client` handler contains 267 eBPF instructions, but due to its loops and branches the verifier evaluates 217,732 instructions. Although `Client` is lower than the verifier's 1m instruction limit, more branches or loop iterations exponentially increases the verifier's work.

Figure 5 shows Tigger's hybrid design and steps through the user-space and eBPF features that enable user-bypass. After Tigger loads its handlers (i.e., `Client` and `Server`) and maps into kernel-space, ❶ its user-space component opens and authenticates connections to the back-end DBMS for pooling. Next, ❷ Tigger adds the server sockets to `ServerSocketsMap`. This step ensures that the `Server` handler runs whenever a DBMS socket buffer is ready for processing. With the back-end socket ready to accept queries, ❸ Tigger adds it to the stack map of idle sockets. ❹ Tigger resets the state metadata associated with the socket. Upon a new connection request, ❺ the client authenticates with Tigger's user-space component. ❻ Tigger adds the client's socket to `ClientSocketsMap`. The `Client` handler will now execute on buffer activity from a front-end connection. Lastly, ❼ Tigger resets the metadata associated with the client socket stores in `SocketStatesMap`. To apply user-bypass to a different system, developers reproduce this logic in the user-space components of another DBMS proxy.

4.2 DBMS Protocol Logic

When a socket buffer arrives, Tigger's `Server` and `Client` handlers perform DPI to apply DBMS protocol logic — extracting the state of client sessions to implement features like connection pooling.

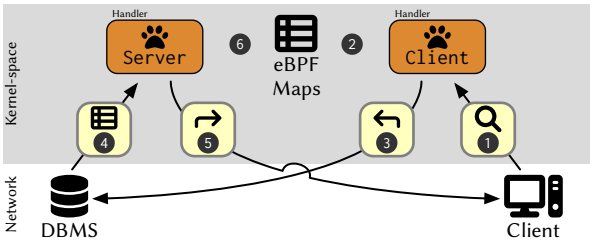


Figure 7: Connection Pooling with User-Bypass – Tigger performs pooling with two handlers and the maps shown in Figure 5. We describe the steps in Section 5.1.

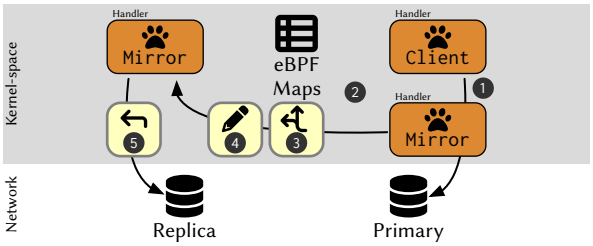


Figure 8: Workload Mirroring with User-Bypass – Tigger performs workload mirroring with additional eBPF programs and maps, including one attached at the TC layer. We describe the steps in Section 5.2.

For PostgreSQL messages, processing a socket buffer involves inspecting each message header to determine its type, length, and, if necessary, the body. A PostgreSQL message is not guaranteed to fit within a single buffer, so `SocketStatesMap` maintains metadata to help the handlers process messages that span buffers. First, it contains enough space to store a partial header in case the header spans multiple socket buffers. Second, it has an offset into the following buffer to find the next message.

Figure 6 shows an example of how Tigger processes multiple PostgreSQL messages in a socket buffer containing 500 bytes. ❶ The `Client` handler reads the first message header and length and computes the location of the next header. ❷ The `Client` handler repeats the process and arrives at the third PostgreSQL message. ❸ The header indicates that the `INSERT` statement is `1213` bytes long, but the socket only has `398` of its `500` bytes remaining. In this scenario, ❹ `Client` stores the offset (i.e., `815`) to look for the following PostgreSQL message header in `SocketStatesMap`. When the next client socket buffer arrives, the handler starts processing at that offset rather than inspecting every byte.

Both the `Client` and `Server` handlers process the PostgreSQL protocol similarly but apply different control logic. `Client` looks for message headers related to the session (e.g., authentication, disconnect). As described in Section 4.1, client messages are typically query requests, and `Client` redirects those to a back-end DBMS. If the buffer contains session messages, `Client` passes it to Tigger's user-space component. Similarly, `Server` looks for control messages that denote transaction status (i.e., active vs. idle), which requires reading the message body and the headers. Based on this information, `Server` uses eBPF maps to coordinate transaction pooling among multiple front-end and back-end connections.

5 TIGGER FEATURES

We now describe how Tigger implements DBMS-specific logic to support the two most important features of proxies: pooling and replication. Although the message protocol will differ for other DBMSs, the way Tigger achieves user-bypass for these features would be the same. Specifics related to the PostgreSQL protocol could be adapted to other DBMS protocols.

5.1 Connection Pooling

As introduced in Section 2.1, connection pooling is when a proxy shares a single server connection across one or more client collections. Tigger supports two common forms of pooling: session and transaction pooling. These settings determine how long a front-end client holds a pooled back-end connection. Session pooling allocates a pooled connection for the duration of a client's session, so it is impossible to multiplex connections to reduce the peak number of connections to the DBMS. This mode requires less work from the DBMS proxy than transaction pooling since it does not maintain transaction state. Instead, the proxies only need to check for messages that terminate the client's session. In contrast, transaction pooling releases connections back to the pool at the end of a transaction, reducing the number of connections to the back-end DBMS as shown in Figure 1. Tigger's `Client` handler does not link clients and servers at authentication time, instead waiting until a query arrives. This approach minimizes how long a client holds a connection and makes it available to other client requests as soon as possible.

Figure 7 details Tigger's steps when handling a client request over a pooled connection. When an authenticated client submits a query, ❶ Tigger's `Client` handler executes when the buffer arrives at the socket layer of the proxy. ❷ `Client` first checks `SocketStatesMap` to see if this socket is already linked to a back-end DBMS socket. If not, Tigger acquires the first socket available from `IdleSocketsMap`. If there are no available user-bypass sockets, the session passes to user-space to be handled by Tigger's slow path. After matching with a user-bypass socket, `Client` processes the buffer (as described in Section 4.2) to determine if it is a session, and if it is `Client` passes it to user-space.

In the typical case the socket buffer contains a query, so `Client` redirects the buffer to the linked user-bypass socket and updates the metadata in `SocketStatesMap`. ❹ The back-end DBMS executes the query and sends the results back to Tigger. The `Server` handler runs on buffer arrival, finding the linked front-end socket for the back-end. ❺ `Server` processes the buffer, stores any intermediary state in `SocketStatesMap`, and redirects the buffer to the linked DBMS socket. ❻ occurs depending on the proxy's pooling mode: at transaction completion for transaction pooling or client disconnect for session pooling. During this step, `Server` unlinks the client from the DBMS and inserts the back-end socket into `IdleSocketsMap`.

5.2 Workload Mirroring

DBAs also deploy DBMS proxies to provide different forms of replication. Applications use such replication for load balancing, high availability, or test environments. One important type of replication is *workload mirroring*, where the proxy sends the same queries to multiple DBMSs but treats one as the authoritative primary node [81]. With mirroring, there are no consensus protocols, result

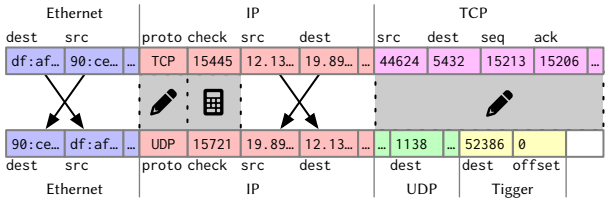


Figure 9: Workload Mirroring Header Manipulation – `Mirror` clones buffers at the TC layer, then manipulates protocol headers to send them back to the socket layer via UDP.

set validation, or awareness between DBMS nodes of their arrangement. It is helpful for prewarming replicas before adding them to the pool of active instances, as well as facilitating the testing of DBMS versions using live traffic during upgrades [58].

Tigger supports workload mirroring between multiple back-end DBMSs. Tigger still performs connection pooling with the primary DBMS, as described in Section 5.1. Replication requires an additional handler (`Mirror`) to send one inbound message to multiple backends. Tigger cannot perform workload mirroring in the `Client` handler due to a limitation in the eBPF verifier: the API to clone socket buffers is unavailable at the socket layer. eBPF programs can only access the clone API at the TC interface.

As shown in Figure 8, Tigger's `Mirror` handler consists of multiple eBPF programs. The first program clones the necessary buffers and attaches as a TC classifier for egress traffic — executing after the socket, TCP, IP, and Ethernet stacks. The second eBPF program attaches at the sockmap layer like the `Client` and `Server` handlers and sends cloned buffers to their replicas. These two programs cooperate in mirroring outbound traffic to replica DBMSs.

When workload mirroring is enabled, Tigger's user-space component adds entries in a eBPF map (`MirrorSocketsMap`) that associates primary DBMS sockets to their replicas' sockets. The user-space component also creates a separate pool of connections for replicas but does not place them into the `IdleSocketsMap` to avoid linking them directly to clients.

Step ❶ in Figure 8 picks up after ❸ in Figure 7 (see Section 5.1). At this point, `Client` bypassed user-space and redirected a query from the client to the primary DBMS. Any new messages arriving at `Client` now respond with "not ready" until all backends are ready. As the outbound message leaves the proxy for the primary, ❷ `Mirror` checks the destination port in `MirrorSocketsMap`, and then ❸ clones the buffer to be sent to a replica.

However, the replica's buffer contains Ethernet, IP, and TCP headers for the primary's session. At the TC layer, manually changing the cloned buffer's headers is not possible: permuting the headers requires Tigger to maintain its own TCP state machines for replicas. Not only is this too complex to implement in eBPF, it is also impossible to use the OS's network stack for any further communication on that socket due to mismatches with OS's TCP logic.

To overcome this problem, `Mirror` uses a eBPF program at the sockmap layer to redirect the cloned buffer to the replica, similar to `Client` and `Server`. This program allows the OS to manage all communication with replicas, but the cloned buffer still resides at the TC layer. To get it back to the `Mirror` handler at the sockmap layer, ❹ `Mirror`'s TC program manually changes the cloned buffer's message type from egress TCP to ingress UDP, as shown in Figure 9.

Table 1: AWS EC2 Instance Details – Hardware configurations and pricing for the c6i family of AWS EC2 instances.

Instance	vCPUs	RAM (GB)	Cost ($/hr)
c6i.large	2	4	0.085
c6i.xlarge	4	8	0.17
c6i.2xlarge	8	16	0.34
c6i.4xlarge	16	32	0.68
c6i.8xlarge	32	64	1.36
c6i.12xlarge	48	96	2.04

To change all the necessary headers, `Mirror` first swaps the Ethernet header's addresses, which does not change the header's checksum. Similarly, `Mirror` switches the IP header's addresses and changes the protocol to UDP. The latter operation requires updating the IP header checksum, but the change from TCP to UDP is a compile-time constant, so this is a fast operation. The final header update completely overwrites the old egress TCP header with an ingress UDP header. `Mirror`'s TC component then writes two pieces of metadata in the buffer after the UDP header. Since TCP headers are larger than UDP, Tigger uses these unused bytes to store (1) the replica's socket for this buffer and (2) the offset to application data. TCP headers are variable length, so Tigger must explicitly track where the DBMS message begins.

When the cloned buffer arrives at the socket layer, ❺ `Mirror` extracts the stored replica socket and trims the excess bytes from the original TCP header. Keeping the destination socket in the buffer is an optimization that enables `Mirror`'s sockmap program to redirect buffers without retrieving additional map data. Lastly, `Mirror` redirects the buffer to the replica DBMS, and updates `SocketStatesMap` so that `Client` can synchronize with all the responses.

6 EVALUATION

We evaluate Tigger's using PostgreSQL (v14.5) DBMS, and configure the `shared_buffers` knob so that working sets fit in memory. We compare Tigger against three other open-source PostgreSQL-compatible proxies:

- **PgBouncer (v1.17):** With decades of development, PgBouncer is the most popular proxy for connection pooling with PostgreSQL. Due to its popularity and maturity, we use PgBouncer as the reference implementation for user-space DBMS proxies.
- **Odyssey (v1.3):** Yandex developed Odyssey as a modern replacement for PgBouncer. It uses multiple workers and coroutines to support parallelism across connections. We use Odyssey as an example of a high-performance user-space DBMS proxy.
- **Pgpool-II (v4.3.3):** This proxy predates PostgreSQL's native replication features, and was commonly deployed to provide high availability for PostgreSQL clusters. Pgpool-II does not support transaction pooling, so we omit it from most of the evaluation. Instead, we only compare Pgpool-II's workload mirroring against Tigger's since PgBouncer and Odyssey do not support that feature.

We do not compare against RDS Proxy because it is a fully managed service. This design makes it impossible to control RDS Proxy's instance size or investigate its performance characteristics.

Unless otherwise specified, all experiments run as follows. We evaluate Tigger using AWS EC2 c6i instances running Ubuntu Linux

Table 2: Connection Pooling for Many Clients – Transaction latencies (ms) of DBMS proxies compared to connecting directly to PostgreSQL.

	No-op			SmallBank			TATP			TPC-C			Twitter			YCSB		
	$\bar{x}$	p50	p99	$\bar{x}$	p50	p99	$\bar{x}$	p50	p99	$\bar{x}$	p50	p99	$\bar{x}$	p50	p99	$\bar{x}$	p50	p99
No proxy	0.18	0.15	0.32	2.10	1.84	4.14	0.90	0.48	2.60	13.92	5.74	224.84	0.72	0.47	2.07	0.62	0.36	1.72
PgBouncer	0.34	0.32	0.52	2.16	2.16	3.58	0.84	0.63	2.16	19.55	7.44	339.95	0.66	0.62	1.48	0.50	0.47	0.90
Odyssey	0.35	0.28	0.48	2.19	2.12	3.38	0.88	0.59	2.13	52.24	7.76	1259.11	0.59	0.48	1.29	0.60	0.46	1.08
Tigger	0.24	0.22	0.39	1.96	1.99	3.20	0.71	0.50	1.96	16.08	7.15	258.00	0.52	0.48	1.29	0.40	0.37	0.76

22.04 LTS in the same availability zone [1]. Table 1 summarizes the resources and pricing for these instances. We use separate instances for each system component: (1) 12xlarge for PostgreSQL, (2) 12xlarge for application servers, and (3) and xlarge for DBMS proxies. We use a smaller instance for the proxies to reflect how users provision proxies in real-world deployments. To better compare user-bypass, we reduce the proxy server's number of receive queues to one to eliminate kernel parallelism for network processing [4].

Each experiment runs at least five times. Latency evaluations use a fixed submission rate of 2000 transactions per second (TPS). This setup ensures all systems perform the same work to compare latency and CPU efficiency. Throughput evaluations run with an unlimited submission rate.

In all box plots, the lower whisker shows the minimum data point, and the upper whisker shows the 99th percentile (p99) data point. We plot p99 instead of the maximum for two reasons: (1) p99 is a standard metric when evaluating the latency of software systems, and (2) write-heavy workloads generate latency outliers due to DBMS resource conflicts (e.g., locks, `fsync()`) that do not reflect the performance of DBMS proxies.

6.1 Workloads

In these experiments, we only consider OLTP workloads; OLAP workloads (e.g., TPC-H) are bottlenecked by query execution at the DBMS (e.g., scans, aggregations), which makes them unsuitable for demonstrating the benefits of Tigger's user-bypass design. In a serverless OLAP scenario, connection establishment overhead will not be the bottleneck compared to query execution. Customers deploy DBMS proxies for OLAP for reasons other than performance (e.g., security) outside this evaluation's scope.

All queries execute over JDBC using the BenchBase framework [8, 43]. We turn off automatically prepared statements in the JDBC driver to avoid naming contamination when the back-end connections are shared across client connections. This problem occurs when drivers prepare different statements under the same name, so DBAs turn off this feature when deploying a DBMS proxy.

- **No-op:** The workload contains a single transaction that executes an empty query string (i.e., ";"). The DBMS returns an empty query result. This workload is ideal for measuring DBMS proxy overhead because it minimizes the work that the client and DBMS execute.
- **SmallBank:** This workload models a banking application where transactions perform short read and update operations [30, 35]. The database contains three tables which we scale to ~3.4 GB.
- **TATP:** This benchmark simulates a cellphone caller location system [84]. It has nine transaction types, and we scale the database to ~1.9 GB stored across four tables.

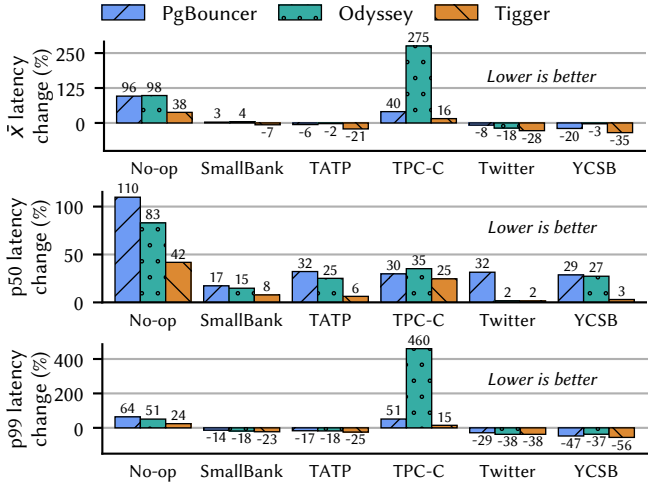


Figure 10: Connection Pooling for Many Clients – Percent change of Table 2 comparing different DBMS proxies against connecting directly to PostgreSQL.

- **TPC-C:** This order-processing application contains nine tables and five transaction types [79]. For our experiments, we use a 20-warehouse database (~2.1 GB).
- **Twitter:** This workload models the troubled social media website where users post messages and follow others. We scale the database to ~4.4 GB.
- **YCSB:** The Yahoo! Cloud Serving Benchmark models cloud service workloads [40]. We run read-only transactions to reduce bottlenecks from EBS writes. The database contains a single table, 1 KB tuples, and a total database size of ~2.3 GB.

6.2 Connection Pooling for Many Clients

We begin our evaluation by exploring the scenario from Section 2.1 in more detail. The workload represents a typical cloud-native application. We use 25 application servers, each with 400 clients, for 10,000 database connections. The DBMS utilization is low, with each of the 25 application servers submitting 80 TPS distributed across their connections, for a total of 2,000 TPS arriving at the back-end DBMS.

Table 2 shows summary statistics of transaction latencies for the six workloads. To more easily quantify overheads and benefits, Figure 10 shows the percent change for each DBMS proxy compared to the scenario with no proxy. Due to its user-bypass design, Tigger offers the lowest latencies of any DBMS proxy in every workload. Compared to running without a proxy, Tigger lowers the mean and p99 latencies in every workload except No-op and TPC-C.

No-op does not benefit from using a DBMS proxy. Even at a high connection count, PostgreSQL performs minimal work for No-op,

Table 3: Connection Pooling for Many TPC-C Clients – Transaction latencies (ms) of DBMS proxies compared to connecting directly to PostgreSQL.

	Delivery			NewOrder			OrderStatus			Payment			StockLevel		
	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99	$\overline{x}$	p50	p99
No proxy	8.68	8.02	26.42	9.57	7.55	42.99	1.85	1.22	7.31	21.05	3.25	562.54	3.60	3.08	6.36
PgBouncer	14.04	10.84	143.69	14.15	9.82	157.60	4.71	1.42	128.15	28.41	3.69	756.52	5.42	2.22	128.59
Odyssey	36.47	11.57	824.07	39.52	10.28	887.05	26.97	1.44	824.11	71.46	3.79	1578.33	29.43	2.25	881.82
Tigger	10.42	10.15	24.04	10.85	9.36	31.28	1.46	1.26	3.77	24.77	3.54	691.03	2.27	2.06	4.53

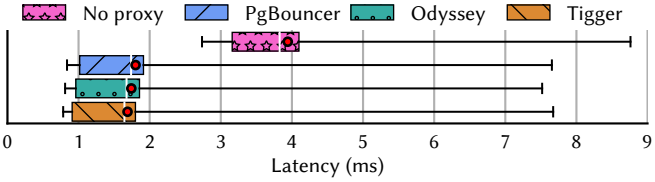


Figure 11: Connection Pooling for Serverless Clients – YCSB transaction latencies showing the effect of proxies' persistent connections. The red circle shows sample mean, and the upper whisker shows p99.

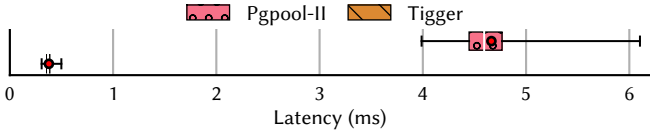


Figure 12: Workload Mirroring – Transaction latencies through DBMS proxies configured for workload mirroring. The red circle shows sample mean, and the upper whisker shows p99.

so reducing the number of connections with a DBMS proxy provides no benefit. However, as a microbenchmark for proxies' protocol efficiency, we see that Tigger offers up to 31% lower latency than the other proxies. The size of the ingress query and egress results are essentially zero, so we are seeing the benefits of user-bypass eliding system calls and user-space thread scheduling.

The diversity of TPC-C's transactions makes it difficult to summarize its performance, as in Table 2. For example, Delivery includes more queries and thus round-trips to the DBMS. OrderStatus and StockLevel are short and read-only. NewOrder and Payment experience more contention, especially when the number of clients exceeds the number of warehouses. These characteristics and non-uniform transaction profile submission rates result in a multimodal latency distribution. For that reason, we provide per-transaction summary statistics.

Table 3 shows that while Tigger reduces the p99 latency in four out of five TPC-C transactions, Payment's higher submission rate and latency hide this in Table 2. Tigger's user-bypass design offers the best performance of the three proxies again, though we cannot explain Odyssey's poor performance. TPC-C's performance subtleties demonstrate the trade-offs in deploying a DBMS proxy. For example, deploying Tigger may be a good decision if reduced tail latencies are worth the slight increase in average latency.

6.3 Connection Pooling for Serverless Clients

We next revisit the YCSB experiment from Section 2.2 that evaluates the latency impact of short-lived connections — this time with more proxies than Tigger. We aim to measure the proxies' connection creation overhead and assess the benefits of Tigger's user-bypass design. Because BenchBase opens a connection at the start of a transaction and closes the connection at the end, we configure the proxies to use session pooling for this experiment. Thus the results from Figures 3 and 11 are not directly comparable to other transaction pooling experiments. The effect is the same (i.e., connections are released to the pool at the end of the transaction), but this configuration requires the proxies to perform less work to track transaction status.

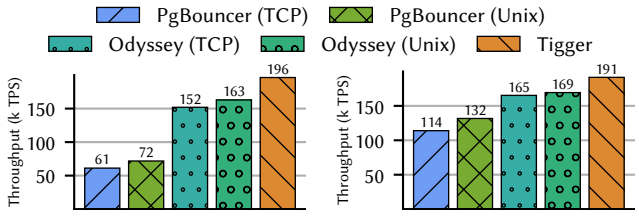
Figure 11 shows the same data as Figure 3 but with latency measurements for Odyssey and PgBouncer. As discussed in Section 2.2, DBMS proxies provide performance improvements in serverless environments by maintaining a connection pool. Figure 11 shows how allocating a connection from a persistent pool is about 2 ms faster than going to the DBMS for a new connection.

Tigger outperforms the other two proxies, offering the lowest minimum, mean, and median latencies. Tail latency is slightly higher for Tigger because this scenario is ill-suited to Tigger's user-bypass design. As described in Section 4.1, Tigger passes client authentication to its user-space components, which ultimately do more work than PgBouncer's because Tigger has to maintain metadata in eBPF maps. Tigger handling subsequent queries with user-bypass makes up for the authentication overhead to provide the best performance on average for serverless clients.

6.4 Workload Mirroring

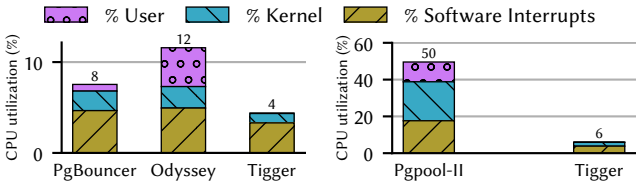
We now evaluate Tigger's workload mirroring as described in Section 5.2. For this experiment, we deploy PostgreSQL with two nodes: (1) primary and (2) replica. We configure Pgpool-II [20] to use its "native replication" clustering mode that routes queries to connection pools for both the primary and the replica DBMSs. The proxy returns results to the client from the primary, and there is no validation that both servers produced the same results. The proxy only ensures that both servers are ready for the following query before allowing the client to proceed. This scenario mimics simultaneously sending an actual workload to a production and staging DBMS. We configure Tigger's mirroring logic to behave like Pgpool-II. Pgpool-II does not support transaction pooling, so we use session pooling and lower the number of client connections to 20.

Figure 12 shows the mean latencies of YCSB in the mirroring scenario described above. On average, Tigger produces transaction latencies that are 92% lower than those with Pgpool-II. Mirroring is expensive to implement in user-space because for each replica, a user-space proxy must copy the queries and results from the kernel, which is the most costly part of the Linux networking stack. In contrast, Tigger's user-bypass design benefits from eliding system calls and zero-copy socket buffer duplication in the Linux kernel to enable mirroring. Comparing Tigger's performance from another YCSB experiment in Table 2 without mirroring and much more connections, we see that Tigger maintains <1 ms transaction latency in both scenarios.



(a) Transaction pooling **(b)** Session pooling

Figure 13: Protocol Efficiency – No-op throughput to compare transaction against session pooling, and TCP against Unix sockets.



(a) Transaction pooling **(b)** Workload mirroring

Figure 14: CPU Efficiency – Breakdown of CPU utilization for YCSB in Table 2 and Figure 12. % Software Interrupts is distinct from % Kernel to emphasize Tigger's user-bypass execution.

6.5 Protocol Efficiency

We next evaluate the proxies' efficiency at applying DBMS protocol logic and explore the overheads associated with TCP and Unix sockets. We configure the proxies for both session and transaction pooling, as detailed in Section 5.1. We reduce the number of terminals to 20 because we are using session pooling and do not want the number of back-end DBMS connections to get too large. This experiment evaluates how efficiently proxies apply knowledge of the DBMS's network protocol to find transaction boundaries in messages and maintain state about connection status.

To avoid unnecessary overhead from the DBMS and exercise proxies' logic as much as possible, we run the No-op workload. We also run the proxies on the same server as the DBMS to reduce network hops. This configuration also allows us to evaluate Odyssey and PgBouncer with TCP and Unix socket connections to the DBMS. Unix sockets are more efficient than TCP networking, so we enable this optimization to quantify their benefit.

The results in Figure 13a show the throughput of the proxies in transaction pooling mode. As a baseline, No-op throughput when BenchBase connects to PostgreSQL without a proxy is 214k TPS. PgBouncer performs the worst, with 71% and 67% reductions in No-op throughput over TCP and Unix sockets, respectively. Odyssey's throughput drops 29% and 24% over TCP and Unix sockets. Unix sockets perform better for both proxies because it is a more efficient form of interprocess communication than going through the entire TCP stack. In comparing these results to Figure 13b, we see that Tigger experiences no throughput degradation when performing extra logic for transaction pooling. The other proxies do not perform as poorly with the simpler task of session pooling, but Tigger offers consistently better performance and pushes the bottleneck to communication overhead elsewhere.

These results also show the benefit of Tigger's user-bypass, as it outperforms the other proxies, reducing the throughput by only 8% compared to the baseline performance. No-op's messages are small (i.e., fewer than 64 bytes), so the primary benefit of user-bypass is reducing system call overhead. Unix sockets between the DBMS and proxy yield little benefit over TCP sockets.

6.6 CPU Efficiency

We next evaluate the CPU efficiency of Tigger's user-bypass approach compared to other DBMS proxies using YCSB. We measure the proxy server's CPU utilization using `sysstat` [13], which attributes time spent in user-space, kernel-space, and software interrupts. We are interested in the latter because Tigger's eBPF programs attached at the sockmap layer run on kernel threads queued as software interrupts (see Section 4.1).

Figure 14a shows the CPU utilization for the YCSB experiment in Table 2. PgBouncer again serves as the baseline. Although Odyssey provides lower latencies than PgBouncer, it increases the CPU overhead by 54%. Odyssey's complex user-space logic for coroutines incurs high overhead to perform the same amount of work as PgBouncer. Lastly, Tigger requires the fewest CPU cycles while providing the best performance of the three proxies. As expected, Tigger's user-bypass design shows almost no CPU time spent in user-space, and reduces the time spent in the kernel by not repeatedly copying socket buffers in system calls.

Figure 14b shows the CPU utilization for the YCSB experiment in Figure 12. Tigger uses 88% less CPU than Pgpool-II to perform workload mirroring. Pgpool-II must copy buffers and duplicate system calls to perform mirroring, which incurs significant overhead. Tigger benefits from zero-copy socket buffer cloning and redirects these buffers to mirrors without system calls.

6.7 Large Data Transfer

Although large data transfers through DBMS proxies are not as common as transactional workloads, we evaluate the overhead of performing such a task. In this scenario, we run the `pg_dump` utility that creates a backup of a PostgreSQL database over the network. `pg_dump` connects to the DBMS like any client, communicates over the PostgreSQL protocol, and extracts database contents data using the `COPY` SQL command. We run `pg_dump` on a dedicated application server, create a PostgreSQL database with 20 tables, and insert 50m single-column `INTEGER` tuples into each table. The overall database size is 34 GB.

Figure 15 shows the elapsed time for `pg_dump` to complete a backup in different proxy scenarios. All three proxies complete the copy without overhead (i.e., `pg_dump`'s efficiency is the bottleneck). This experiment demonstrates a workload where the bottleneck exists at the client or the DBMS, rather than the proxy. In this scenario, a DBMS proxy adds no measurable overhead.

6.8 Proxy Instance Size

To complete our evaluation, we now measure the maximum throughput of the proxies with varying EC2 instance sizes. While customers typically deploy proxies on weak servers, we aim to investigate their performance benefits when allocated more resources. For Odyssey and Tigger, this offers opportunities to express parallelism, while PgBouncer only supports a single worker. We run No-op, TPC-C, and YCSB with unlimited submission rates.

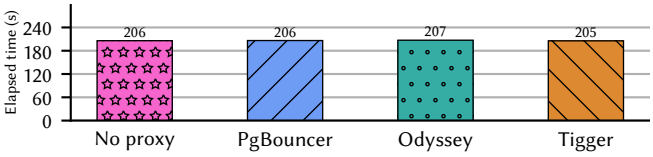


Figure 15: Large Data Transfer – Elapsed time for `pg_dump` to back up a 34 GB database through different DBMS proxies.

Figure 16 shows the throughput of the different DBMS proxies on varying instance sizes. For every instance type, Tigger yields the highest throughput. All three proxies scale roughly linearly as the resources of each instance increase. For Odyssey, this makes sense because it uses multiple user-space worker threads, so Odyssey expresses greater parallelism. PgBouncer performs better as instance types change despite being single-threaded. We attribute this to the noisy neighbor problem: the more resources requested from a multi-tenant system, the less competition for unprovisioned hardware (i.e., CPU caches).

With No-op and the smallest instance type (i.e., large, which the other experiments use for the proxy), Tigger more than doubles the throughput of PgBouncer and Odyssey. In this scenario, all three proxies are bottlenecked by CPU utilization. Tigger's higher performance demonstrates the benefit of user-bypass, which does not waste CPU cycles shuffling network buffers between user-space and kernel-space. In all three workloads, Odyssey requires an 8xlarge instance at 8× the cost to match Tigger's baseline performance on a large instance. This experiment demonstrates the benefit of user-bypass to efficiently use CPU resources in resource-constrained scenarios and the impact that user-bypass has on cloud cost.

7 RELATED WORK

We now discuss existing research on eBPF observability, programmable networks, and techniques similar to user-bypass for pushing DBMS-specific logic into kernel-space.

Network Observability Proxies: Envoy is an L3/L4 proxy with some L7 capabilities for DBMS observability [22]. Endpoint support is limited to HTTP/2 and gRPC, along with "sniffing filters" for DBMSs like MongoDB and PostgreSQL. For PostgreSQL, Envoy can accumulate counters for queries (i.e., `INSERT`, `UPDATE`) and errors in the query stream but does not perform connection pooling or workload replication. Because it is not a PostgreSQL endpoint, it cannot process encrypted traffic.

Network Function Virtualization (NFV): Custom behavior in the network stack and middleboxes has increased in recent years, particularly with the rise of SmartNICs among cloud vendors [47]. Programming languages like P4 [34] and Domino [73] allow more expressibility in the network layer, but developers apply them to flow routing using packet headers rather than L7 logic.

Recent works increasingly rely on the benefits of eBPF to push logic into kernel-space and even to hardware layers [45, 65, 68, 82, 86]. These eBPF efforts target L3/L4 applications like flow routing, intrusion detection, and denial-of-service mitigation and rely on a class of eBPF programs called XDP which are more restrictive than Tigger's sockmap programs. BMC also uses XDP to apply user-bypass to memcached, enabling kernel threads to handle the key-value store's simple operations over UDP [51].

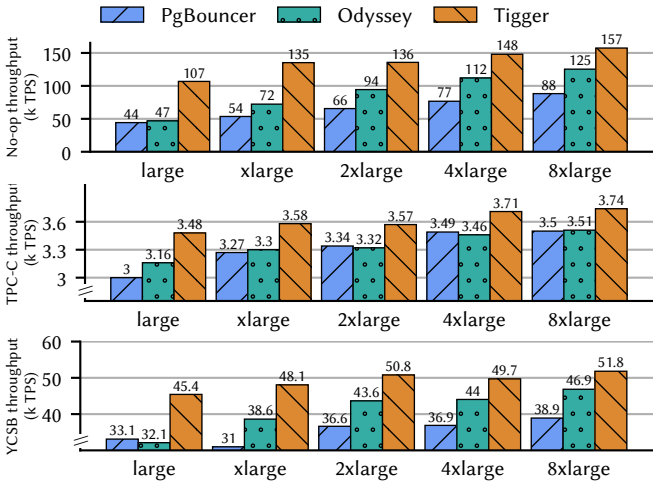


Figure 16: Proxy Instance Size – Transaction throughput while varying proxy server instance size.

TCP Splicing: Network researchers explored kernel-space connection pooling at the transport layer (e.g., TCP) in kernel-space with a technique called *TCP splicing* [62]. The goal was similar to user-bypass: avoid copying buffers to user-space that will ultimately be forwarded to another socket. Prior work evaluated these techniques on web server and firewall workloads but required invasive changes to the OS to implement. For example, IBM AIX developers made changes directly to the kernel source code [67], a Linux implementation required an out-of-tree kernel module [39], and other approaches used custom designs like Scout OS and Exokernel [50, 75]. The Linux effort was unique in applying L7 logic to provide URL-aware forwarding [39]. This logic could be duplicated using user-bypass to offer a safe implementation without risking kernel panics.

One outcome of these efforts is the `splice()` system call in Linux that promises zero-copy forwarding between file descriptors without duplicating the data in user-space. However, this approach still incurs system call overhead and requires waking up an user-space application to coordinate forwarding. Lastly, it is impossible to peek at the data in flight, so DPI to apply DBMS logic requires copying the data to user-space.

OS Extensibility: For decades, database researchers discussed the shortcomings of using OS services to design DBMSs [78]. In response, the research community proposed methods to extend OS behavior to better suit user-space applications [32, 69, 70]. However, these approaches lacked a standard API and strong OS support like eBPF to extend behavior.

More recently, researchers presented the ExtOS Linux prototype designed to reduce data movement in the software stack [31]. For example, they proposed adding programmability to the `read()` system call to push database filters into the OS via a kernel module. Their initial results showed promising speedup, and they proposed extensions to eBPF which was not yet suitable for their task in 2019.

Unikernels: These applications represent kernel-bypass pushed to an extreme, where applications are compiled with a library OS to yield an application-specific machine [46, 55, 59, 83]. Unikernels remain an active and promising research area, but the industry has

been slow to adopt this approach. Designing unikernel applications requires similar expertise and has the same challenges as kernel-bypass methods. While more limited in capabilities, eBPF is easily deployed in cloud native environments and companies are rapidly adopting it, as described in Section 3.2.

eXpress Resubmission Path (XRP): Recent work applies user-bypass to the Linux kernel's storage stack [87]. For example, B+ tree lookups read multiple disk pages before finding the destination leaf node. The DBMS copies the entire page from kernel-space to user-space via system calls, only to find a pointer to the correct child node and repeat the process. Similar to the network stack, repeatedly accessing the storage stack imposes overhead from system calls and memory copies.

XRP pushes DBMS logic into the NVMe driver via eBPF. With XRP, the DBMS performs B+ tree node traversal in kernel-space by resubmitting multiple NVMe operations. XRP's use of user-bypass reduced the amount of data copied to user-space and the number of repeated system calls. They demonstrated the latency and throughput benefits of XRP against kernel-bypass using DPDK and asynchronous I/O.

Kernel-Bypass: As discussed in Section 3.1, kernel-bypass is a technique to bring packets directly to user-space by eliding the Linux network stack. We created a version of PgBouncer that uses kernel-bypass via F-Stack [5] to evaluate the benefits and challenges of kernel-bypass, especially compared to user-bypass. But integrating F-Stack dependencies and creating a productive development environment proved challenging. For example, DPDK requires exclusive control of a dedicated NIC, making debugging difficult with standard networking tools. Despite communicating with F-Stack engineers, we did not achieve acceptable performance with DPDK-enhanced PgBouncer; it is 10–100× slower than unmodified PgBouncer. Our engineering problems match the recent effort to create a DPDK version of Open vSwitch [80], which concludes by recommending AF_XDP instead.

To our knowledge, the only database vendor making significant use of kernel-bypass is Yellowbrick, which does not use DPDK but instead wrote their own network and storage device drivers [53]. Although ScyllaDB promotes DPDK compatibility in their Seastar framework, they do not deploy DPDK in production [52]. In general, deploying DPDK applications into production is difficult due to API/ABI instability, which forces developers to choose between features and fixes of new releases and code stability of LTS releases [64]. eBPF experienced similar API growing pains over the last ten years but is proving to be a less burdensome development environment than kernel-bypass.

8 FUTURE WORK

This work presents a technique called user-bypass in the context of DBMS proxies. We believe this opens a number of opportunities for both DBMS proxies and aspects of DBMS design that could benefit from application logic in kernel-space.

Proxy Features: Tigger implements the most common features needed in DBMS proxies to demonstrate the benefits of user-bypass, but there are more capabilities to evaluate. For example, Vitess [74] supports SQL-aware logic to perform query rewriting and sharding for MySQL. Also, techniques that modify the query stream, like transaction reordering and in-network computation [44, 57], may be feasible in a DBMS proxy using user-bypass.

Pgpool-II and ProxySQL provide a query result cache to return result sets for frequently submitted queries. These caches rely on simple time-to-live (TTL) policies with string-matching of queries rather than SQL-aware eviction methods. For example, a cached result will not automatically be evicted if a query changes its value in the DBMS. For this reason, these caches are error-prone but could be used to reduce the burden of queries automatically submitted by application frameworks like "`SELECT 1`". More sophisticated solutions (e.g., Heimdall [15]) support transparent query caching with automatic invalidation. These techniques could benefit from user-bypass if their approaches satisfy the eBPF verifier. For example, a full SQL parser is outside the verifier's scope, and eBPF maps may not be large enough for a robust query result cache.

Asynchronous I/O: Linux provides an asynchronous I/O interface called *io_uring*. In late 2021, a Linux developer posted an experimental patch for io_uring that offers support for network sockets [41]. The new kernel feature reads and writes via shared buffers with user-space, which reduces the number of data copies. A DBMS proxy would still need to copy data between these buffers, and all coordination would still be performed via user-space application waiting on `epoll()`. Nonetheless, we remain interested in evaluating io_uring's impact on DBMS proxy design.

Hardware Acceleration: The improved capabilities of SmartNICs and FPGAs and their proliferation in public cloud settings create new opportunities for accelerating applications via hardware [47]. For example, Mellanox and Netflix added support for offloading kTLS to NICs [71]. This development may enable performing L7 logic like DBMS proxies at lower levels of the network stack. Hardware TLS is not broadly available, but future devices may enable user-bypass techniques at the hardware layer.

9 CONCLUSION

DBMS proxies manage connections to address scalability and connection life cycle issues introduced by modern cloud applications. These programs apply DBMS-specific protocol logic to multiplex client connections. But these proxies incur inefficiencies because they are user-space applications that rely on system calls to copy buffers to and from kernel-space. We introduce a user-bypass technique to overcome these shortcomings by pushing application logic into the Linux kernel using eBPF. We show how to apply user-bypass with our PostgreSQL-compatible proxy Tigger. When compared against other DBMS proxies, Tigger offers the best performance and lowest operating cost. Our evaluation shows the value of user-bypass to reduce data movement between kernel-space and user-space and avoid costly system calls.

ACKNOWLEDGMENTS

This work was supported (in part) by the National Science Foundation (IIS-1846158, SPX-1822933), VMware Research Grants for Databases, Google DAPA Research Grants, and the Alfred P. Sloan Research Fellowship program.

REFERENCES

[1] [n.d.]. Amazon EC2 C6i Instances - Amazon Web Services. https://aws.amazon.com/ec2/instance-types/c6i/.

[2] [n.d.]. Amazon RDS Proxy | Highly Available Database Proxy | Amazon Web Services. https://aws.amazon.com/rds/proxy/.

[3] [n.d.]. App Scaling - AWS Application Auto Scaling - AWS. https://aws.amazon.com/autoscaling/.

[4] [n.d.]. ENA Linux Driver Best Practices and Performance Optimization Guide. https://github.com/amzn/amzn-drivers/blob/master/kernel/linux/ena/ENA_Linux_Best_Practices.rst.

[5] [n.d.]. F-Stack | High Performance Network Framework Based On DPDK. http://www.f-stack.org.

[6] [n.d.]. GitHub - brettwooldridge/HikariCP: A solid, high-performance, JDBC connection pool at last. https://github.com/brettwooldridge/HikariCP.

[7] [n.d.]. GitHub - cilium/cilium: eBPF-based Networking, Security, and Observability. https://github.com/cilium/cilium.

[8] [n.d.]. GitHub - cmu-db/benchbase: Multi-DBMS SQL Benchmarking Framework via JDBC. https://github.com/cmu-db/benchbase.

[9] [n.d.]. GitHub - facebookincubator/katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran.

[10] [n.d.]. GitHub - haproxy/haproxy: HAProxy Load Balancer's development branch (mirror of git.haproxy.org). https://github.com/haproxy/haproxy/.

[11] [n.d.]. GitHub - mariadb-corporation/MaxScale: An intelligent database proxy. https://github.com/mariadb-corporation/MaxScale.

[12] [n.d.]. GitHub - microsoft/ebpf-for-windows: eBPF implementation that runs on top of Windows. https://github.com/microsoft/ebpf-for-windows.

[13] [n.d.]. GitHub - sysstat/sysstat: Performance monitoring tools for Linux. https://github.com/sysstat/sysstat.

[14] [n.d.]. GitHub - yandex/odyssey: Scalable PostgreSQL connection pooler. https://github.com/yandex/odyssey.

[15] [n.d.]. Home - Heimdall Data. https://www.heimdalldata.com.

[16] [n.d.]. Infrastructure – Vercel. https://vercel.com/features/infrastructure.

[17] [n.d.]. Intel Data Plane Development Kit (DPDK). https://www.dpdk.org.

[18] [n.d.]. nginx. https://nginx.org/en/.

[19] [n.d.]. PgBouncer - lightweight connection pooler for PostgreSQL. https://www.pgbouncer.org.

[20] [n.d.]. pgpool Wiki. https://www.pgpool.net/.

[21] [n.d.]. pgxpool package - github.com/jackc/pgx/v4/pgxpool - Go Packages. https://pkg.go.dev/github.com/jackc/pgx/v4/pgxpool.

[22] [n.d.]. Postgres — envoy 1.24.0-dev-fbcf42 documentation. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_protocols/postgres.

[23] [n.d.]. ProxySQL - A High Performance Open Source MySQL Proxy. https://proxysql.com.

[24] [n.d.]. Seastar. https://seastar.io.

[25] [n.d.]. socket(2) - Linux manual page. https://www.man7.org/linux/man-pages/man2/socket.2.html.

[26] [n.d.]. tc(8) - Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[27] 2021. PHP: Connections and Connection management - Manual. https://www.php.net/manual/en/pdo.connections.php.

[28] 2022. Databases | Django documentation | Django. https://docs.djangoproject.com/en/4.1/ref/databases/.

[29] David Ahern. 2022. Can the Linux networking stack be used with very high speed applications? https://lpc.events/event/16/contributions/1345/.

[30] Mohammad Alomari, Michael J. Cahill, Alan D. Fekete, and Uwe Röhm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE*. IEEE Computer Society, 576–585.

[31] Antonio Barbalace, Javier Picorel, and Pramod Bhatotia. 2019. ExtOS: Data-centric Extensible OS. In *APSys*. ACM, 31–39.

[32] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. 1995. Extensibility, Safety and Performance in the SPIN Operating System. In *SOSP*. ACM, 267–284.

[33] Daniel Borkmann and John Fastabend. 2018. Combining kTLS and BPF for Introspection and Policy Enforcement.

[34] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95.

[35] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *SIGMOD Conference*. ACM, 729–738.

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

[37] Brian E. Carpenter and Scott W. Brim. 2002. Middleboxes: Taxonomy and Issues. *RFC* 3234 (2002), 1–27.

[38] Elizabeth Christensen. 2022. Postgres at Scale: Running Multiple PgBouncers. https://www.crunchydata.com/blog/postgres-at-scale-running-multiple-pgbouncers.

[39] Ariel Cohen, Sampath Rangarajan, and J. Hamilton Slye. 1999. On the Performance of TCP Splicing for URL-Aware Redirection. In *USENIX Symposium on Internet Technologies and Systems*. USENIX.

[40] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.

[41] Jonathan Corbet. 2021. Zero-copy Network Transmission with io_uring. https://lwn.net/Articles/879724/.

[42] J.D. Day and H. Zimmermann. 1983. The OSI reference model. *Proc. IEEE* 71, 12 (1983), 1334–1340. https://doi.org/10.1109/PROC.1983.12775

[43] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[44] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering. *Proc. VLDB Endow.* 12, 2 (2018), 169–182.

[45] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 2019. Partition-Aware Packet Steering Using XDP and eBPF for Improving Application-Level Parallelism. In *ENCP@CoNEXT*. ACM, 27–33.

[46] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP*. ACM, 251–266.

[47] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*. USENIX Association, 51–66.

[48] Andres Freund. 2020. Analyzing the Limits of Connection Scalability in Postgres. https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/analyzing-the-limits-of-connection-scalability-in-postgres/ba-p/1757266.

[49] Andres Freund. 2020. Measuring the Memory Overhead of a Postgres Connection. https://blog.anarazel.de/2020/10/07/measuring-the-memory-overhead-of-a-postgres-connection/.

[50] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Héctor M. Briceño, Russell Hunt, and Thomas Pinckney. 2002. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.* 20, 1 (2002), 49–83.

[51] 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 *NSDI*. USENIX Association, 487–501.

[52] CMU Database Group. 2020. ScyllaDB: No-Compromise Performance (Avi Kivity). https://youtu.be/0S6i9BmuF8U?t=2586.

[53] CMU Database Group. 2022. Yellowbrick: An Elastic Data Warehouse on Kubernetes (Mark Cusack). https://youtu.be/uHMcVDNkHi4.

[54] 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 *CoNEXT*. ACM, 54–66.

[55] Takayuki Imada. 2018. MirageOS Unikernel with Network Acceleration for IoT Cloud Environments. In *ICCBDC*. ACM, 1–5.

[56] Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*. USENIX Association, 489–502.

[57] Theo Jepsen, Alberto Lerner, Fernando Pedone, Robert Soulé, and Philippe Cudré-Mauroux. 2021. In-Network Support for Transaction Triaging. *Proc. VLDB Endow.* 14, 9 (2021), 1626–1639.

[58] Lev Kokotov. 2022. Scaling PostgresML to 1 Million Requests per Second. https://postgresml.org/blog/scaling-postgresml-to-one-million-requests-per-second.

[59] Simon Kuenzer, Vlad-Andrei Badoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Stefan Teodorescu, Costi Raducanu, Cristian Banu, Laurent Mathy, Razvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: fast, specialized unikernels the easy way. In *EuroSys*. ACM, 376–394.

[60] Kahina Lazri, Antoine Blin, Julien Sopena, and Gilles Muller. 2019. Toward an in-Kernel High Performance Key-Value Store Implementation. In *SRDS*. IEEE, 268.

[61] Tim Liang. 2022. The growing pains of database architecture. https://www.figma.com/blog/how-figma-scaled-to-multiple-databases/.

[62] David A. Maltz and Pravin Bhagwat. 1999. TCP Splice for application layer proxy performance. *J. High Speed Networks* 8, 3 (1999), 225–240.

[63] Chris McCord. 2014. Rise of the Phoenix - Building an Elixir Web Framework.

[64] John McNamara, Ian Stokes, Luca Boccassi, and Kevin Traynor. 2017. API/ABI Stability and LTS: Current state and Future. https://www.dpdk.org/event/dpdk-userspace-dublin-2017/.

[65] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *HPSR*. IEEE, 1–8.

[66] Yaser Raja. 2021. Resources consumed by idle PostgreSQL connections | AWS Database Blog. https://aws.amazon.com/blogs/database/resources-consumed-by-idle-postgresql-connections/.

[67] Marcel-Catalin Rosu and Daniela Rosu. 2002. An evaluation of TCP splice benefits in web proxy servers. In *WWW*. ACM, 13–24.

[68] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. 2018. Performance Implications of Packet Filtering with Linux eBPF. In *ITC (1)*. IEEE, 209–217.

[69] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *Proc. VLDB Endow.* 9, 10 (2016), 768–779.

[70] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. 1996. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *OSDI*. ACM, 213–227.

[71] Slava Shwartsman and Drew Gallatin. 2019. Kernel TLS and TLS Hardware Offload. https://papers.freebsd.org/2019/eurobsdcon/shwartsman_gallatin-kernel_tls_harware_offload/.

[72] Giulio Sidoretti, Sebastiano Miano, Stefano Salsano, Gianni Antichi, and Aurojit Panda. 2023. Application Layer Processing Offload in the Kernel. https://2023.eurosys.org/docs/posters/eurosys23posters-final39.pdf. Poster presented at EuroSys 2023.

[73] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *SIGCOMM*. ACM, 15–28.

[74] Sugu Sougoumarane and Mike Solomon. 2012. Vitess: Scaling MySQL at YouTube Using Go. USENIX Association, San Diego, CA.

[75] Oliver Spatscheck, Jørgen S. Hansen, John H. Hartman, and Larry L. Peterson. 2000. Optimizing TCP forwarder performance. *IEEE/ACM Trans. Netw.* 8, 2 (2000), 146–157.

[76] Alexei Starovoitov. 2013. LKML: Alexei Starovoitov [PATCH net-next] extended BPF. https://lkml.org/lkml/2013/9/30/627.

[77] Alexei Starovoitov. 2019. BPF at Facebook. https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/.

[78] Michael Stonebraker. 1981. Operating System Support for Database Management. *Commun. ACM* 24, 7 (1981), 412–418.

[79] The Transaction Processing Council. 2007. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[80] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. revisiting the open vSwitch dataplane ten years later. In *SIGCOMM*. ACM, 245–257.

[81] Marco Tusa. 2017. What About ProxySQL and Mirroring? https://www.percona.com/blog/proxysql-and-mirroring-what-about-it/.

[82] Marcos Augusto M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson Rubens da Silva Santos, Eduardo P. M. Câmara Júnior, and Luiz Filipe M. Vieira. 2021. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1 (2021), 16:1–16:36.

[83] Thiemo Voigt and Bengt Ahlgren. 1999. Scheduling TCP in the Nemesis Operating System. In *Protocols for High-Speed Networks (IFIP Conference Proceedings)*, Vol. 158. Kluwer, 63–80.

[84] Antoni Wolski. [n.d.]. TATP Benchmark. http://tatpbenchmark.sourceforge.net.

[85] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792.

[86] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. 2018. Leveraging eBPF for programmable network functions with IPv6 segment routing. In *CoNEXT*. ACM, 67–72.

[87] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *OSDI*. USENIX Association, 375–393.