Characterizing and Optimizing the Networking Stack in Databases

by

Prabhakar Kafle

B.S. Computer Science and Engineering, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Prabhakar Kafle Department of Electrical Engineering and Computer Science May 17, 2024
Certified by:	Michael Stonebraker Adjunct Professor of CS and Engineering, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Characterizing and Optimizing the Networking Stack in Databases

by

Prabhakar Kafle

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Databases are latency-critical applications, and client-database communication is a significant contributor to the end-to-end latency. However, the database community has paid little attention to the networking overhead in databases. This thesis focuses on the overhead from the network stack in the server. I characterize the contributions of different components in the database server to the end-to-end latency, focusing on the networking stack. I observe that in transactions involving a single read query, the server network stack accounts for almost 15% of the total end-to-end latency in VoltDB. Most of this overhead comes from TCP packet processing, interrupt handling, context switches, and I/O multiplexing. Additionally, this work also explores avenues to optimize the networking stack overhead. I find that moving networking to the userspace by bypassing the kernel can significantly reduce the networking stack overhead. This switch in the network stack can help achieve a significant improvement in throughput and lower latency for both the benchmarks used. While the thesis is focused on server networking stack, similar optimization can be applied to client side if necessary hardware (CPU, NIC) is available.

Thesis supervisor: Michael Stonebraker
Title: Adjunct Professor of CS and Engineering

Acknowledgments

I would like to extend my deepest gratitude to my supervisor, Michael Stonebraker, for his mentorship and guidance throughout my work. Working under him has been a humbling learning experience. I would also like to thank Xinjing Zhou and Darren Lim, with whom I had the pleasure of working. Their help has saved me countless hours and I have learned a lot from their work. Additionally, I am also grateful to Victor Leis and Xiangyao Yu, whose feedback has been helpful in shaping this work. Last but not the least, I would like to thank my family and friends for their support throughout my time at MIT. It would not have been possible without them.

Finally, I would like to acknowledge the use of ChatGPT to proofread the final draft of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the years, there has been a lot of work in the database community to decrease query latency. Almost all these works have focused on making query execution within the database faster.[1][2][3][4][5][6][7] However, there is an often-underlooked component that contributes as much time if not more to the query latency: client-server communication. When a client makes a query request to the database server, the following operations happen sequentially:

(a) Client app serializes the request into bytes

(b) Client app in userspace sends the request bytes to the OS kernel which in turn writes it to the Network Interface Card (NIC) buffer

(c) Client NIC writes request bytes to the network

(d) Data is transferred to the server over the network

(e) Server NIC receives the request and reads the bytes into its buffer

(f) Server NIC triggers interrupt handler. This causes the waiting server OS kernel to read the data from the NIC buffer and pass it to the DB process in userspace

(g) DB processes the query to produce a result

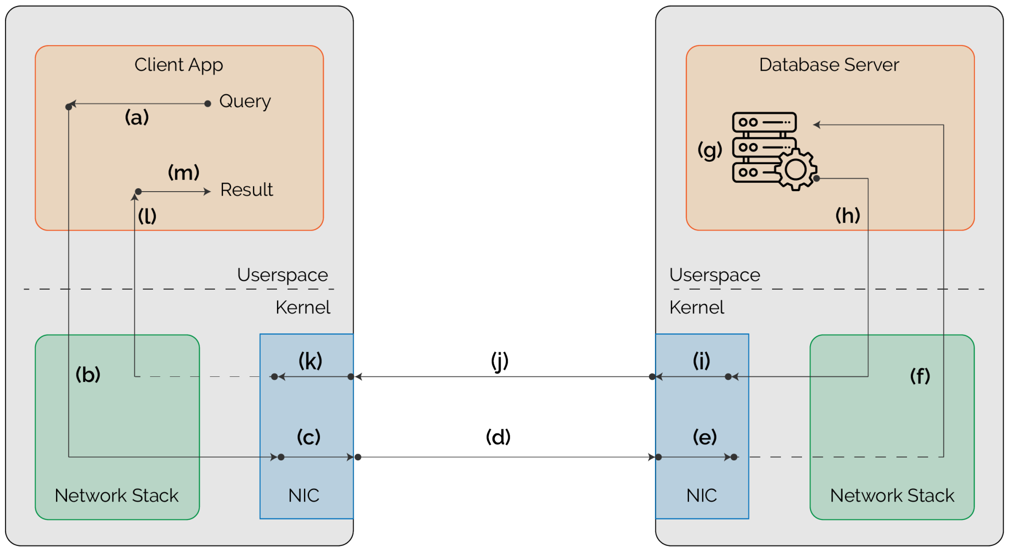


Figure 1.1: Lifespan of a query request from a client to the server and response back to the client

(h) DB passes the result bytes to the OS kernel which writes request bytes to the NIC buffer

(i) Server NIC writes the bytes to the network

(j) Data is transferred back to the client over the network

(k) Client NIC receives the result and reads bytes into its buffer

(l) Client NIC triggers interrupt handler. This causes the waiting client OS kernel to read the data and pass it to the client app in userspace

(m) Client app deserializes bytes and interprets the result.

These steps are visualized clearly in figure 1.1.

Apart from steps (a) and (m) which depend on client implementation, the above steps can be grouped into the following three categories:

(i) **Database implementation-dependent:** Step (g). It depends on the algorithm, language, and other implementation design choices. This is where most of the effort in reducing latency has traditionally gone. It can be further broken down into query planning and execution. For the sake of this thesis, it will be considered as one step.

(ii) **Networking stack dependent:** Steps (b), (f), (h), and (l). These operations are normally handled by the OS drivers and operating system networking stack. This thesis focuses on characterizing the overhead from these operations and proposing optimizations.

(iii) **Network infrastructure dependent:** Steps (c), (d), (e), (i), (j), (k). These depend on the NIC, switches in paths, and other networking infrastructure and variables (like distance between client-server, etc.) These generally cannot be controlled by software in the client or server.

This is an oversimplified view of how a database works. A fully-fledged database has more complexity in thread scheduling, logging, and buffer management. Many of these overheads have been studied before.[8] This work, however, focuses on the networking stack overhead, which has received little attention.

My results show that approximately 50% of the request's lifespan (350us) is spent in the server from the time the query request reaches the server's NIC buffer and when the server writes response to the NIC buffer to send to client i.e. steps (f), (g), and (h) in figure 1.1. Of this time spent in the server, about one-third goes into operating systems-level orchestration to read from and write to the NIC (category ii). This is more than the query execution itself (around 12%). In addition to characterizing the lifespan of a request in the server, this work also explores avenues to optimize the network overhead. As discussed in section 7.1, bypassing the Linux kernel for networking can reduce network overhead from the server stack by up to 80%. Overall, it leads to a 44% increase in throughput and a 28% decrease in latency on the Retwis workload in VoltDB.

The rest of the thesis is structured as follows: chapter (2) discusses the background topics necessary to understand the rest of the thesis; (3) discusses previous literature on the topic; (4) describes an echo server microbenchmark used to isolate and focus only on the networking stack; (5) describes the architecture of VoltDB and the optimization to the database networking stack; (6) describes the experimental setup; (7) presents the results; and (8) is the conclusion and future work.

Chapter 2

Background

I assume the reader knows undergraduate-level computer science and software systems. In this section, I discuss some additional background knowledge needed to understand the rest of the thesis.

2.1 Linux networking stack

Linux follows the 'everything is a file' principle from UNIX.[9] This means that a connection with another device over the network is represented by a file descriptor. So I will use 'socket connection' and 'file descriptor' interchangeably throughout this thesis to refer to the connection between two machines over the network.

While sockets can be both blocking and non-blocking, I will use blocking sockets as an example to explain the Linux networking stack.

Writing data to the socket for transmission is fairly straightforward. One can write the data to the NIC buffer via the kernel. However, receiving packets is not so straightforward since we don't know when the data will arrive. So we need to listen for incoming data. This happens when we call the `read` function. While the networking stack involved in receiving data is complex, we need to know only the following operations to understand the concept of interrupt and the overhead involved when receiving packets:

- After we call `read`, the thread switches from userspace to kernel space and calls `ksys_read`. The caller thread then registers an interest in the kernel to read from the file descriptor and goes to sleep.
- When the data arrives, NIC copies the incoming data to its buffer. This causes a hardware interrupt.
- A background thread called `swapper` (process id 0) handles the hardware interrupt by calling function `asm_common_interrupt` which reads data from the NIC
- `ip_sublist_rcv` and `tcp_v4_do_rcv` are called to process the TCP/IP packets and store them in TCP buffer.
- `swapper` thread wakes up the client thread which then reads the data from TCP buffer by caling `tcp_recvmsg`

Graphs of full call stack involved in receiving packets can be seen in appendix A.2.

2.2 IO multiplexing and `epoll`

Performing a network read as described above means that the program could block on a `read` waiting for a packet to arrive. This is problematic if the program has multiple socket connections, as is common in servers. One approach to overcome this issue is to handle each socket connection in a separate thread. However, this is inefficient, especially if the (server) program needs to handle many clients. A better solution is to use an IO multiplexor. IO multiplexing allows a program to listen to multiple file descriptors simultaneously. So instead of listening to each socket connection separately, a program can instrument an IO multiplexer to keep track of all the socket connections and notify when there is any event of interest in any of them.

`epoll` is the default Linux API for IO multiplexing.

2.3 Network Protocol

The transport protocol sits on top of Internet Protocol (IP) and defines the delivery order, reliability, flow control, and congestion control of packets sent over the network. The choice of transport protocol provides a trade-off between the overhead we are willing to incur and the guarantees/features we want from the protocol. Transport Control Protocol (TCP) and User Datagram Protocol (UDP) are the most common transport protocols.

TCP has become the de facto networking protocol for most purposes due to its reliable in-order transmission guarantees. It achieves this with a combination of packet acknowledgments, sequence numbers, checksum, and automatic retransmission. In addition, it has additional features like congestion control to ensure stable and optimal throughput. However, these features come at a cost. TCP packet processing is known to have significant overhead.[10] Furthermore, TCP adds 20-60 bytes of headers to each packet.[11]

Unlike TCP, **UDP** only adds 8 bytes of headers for the port numbers, data length, and checksum.[12] Thus it allows checking for the integrity of the packets but does not guarantee reliable in-order delivery or provide features like congestion control.

2.4 Kernel Bypass: DPDK and F-stack

Linux networking stack is inefficient for high network load due to the overhead from packet processing, locking, buffer copy, and context switches.[13] This presents scalability problems. One solution to this problem is to bypass the kernel completely for networking. This means running the network stack in userspace and interacting directly with the NIC. This approach follows the end-to-end principle[14] by providing control to the end programs to implement the desired features rather than relying on a lower-level generic implementation in the kernel.

Data Plane Development Kit (DPDK) is an open source[1] toolkit developed by Intel

[1] https://github.com/DPDK/dpdk

that provides a set of libraries and drivers to communicate with the NIC from userspace by bypassing kernel. On top of bypassing the kernel, it uses lockless queues, poll mode drivers, and pre-allocated buffers for fast packet processing.

DPDK operates at the Ethernet level but any communication over the internet requires Internet Protocol (IP) and some transport protocol. For this, I use an open-source TCP/IP library built on top of DPDK called F-Stack.[15] This choice was driven by three major factors:

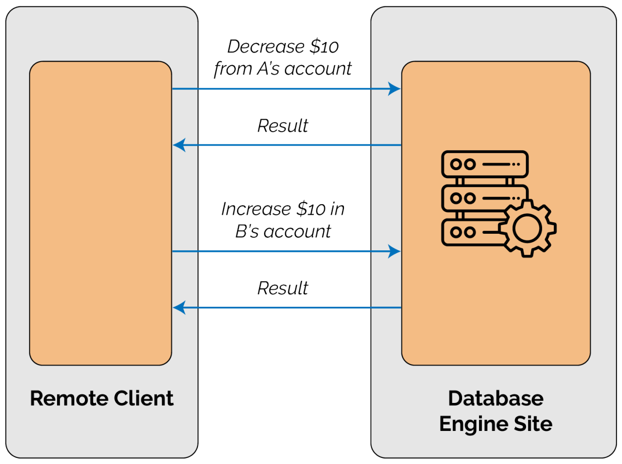
1. F-Stack provides significant improvement to network performance compared to TCP/IP stack in Linux.[16]
2. F-Stack modifies FreeBSD networking code to use DPDK by removing any lock contention. This makes it faster and less error-prone than a new custom TCP/IP implementation.
3. It provides POSIX-like APIs to perform socket-related operations. This makes it easier to port over programs written for the Linux stack.

2.5 perf

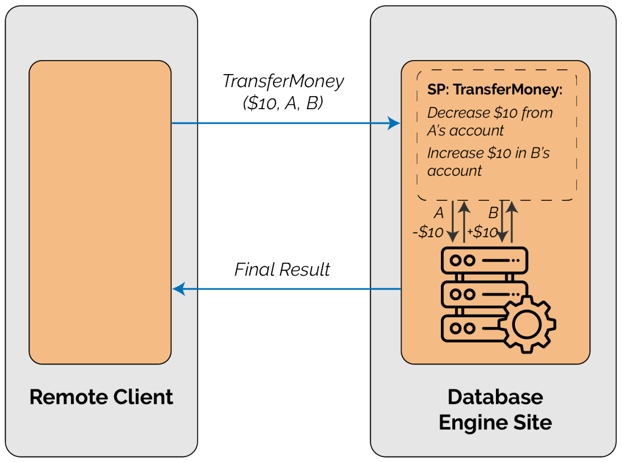
`perf` is a performance analysis tool included in the Linux kernel.[17] It can profile the whole system or a specific process by instrumenting CPU performance counters, tracepoints, and kernel and userspace events with very little overhead.

2.6 Stored procedure

A stored procedure is a group of SQL queries that can be saved in the database and executed with one invocation. Figure 2.1 shows an example comparing interactive query and stored procedure. A stored procedure invocation only requires 1 roundtrip communication between



(a)
Interactive Query. The client sends each SQL query to the server to fetch the result. Each SQL query incurs a round trip to and from the server increasing the latency.


TransferMoney ($10, A, B)
SP: TransferMoney:
Decrease $10 from A's account
Increase $10 in B's account
A -$10
B +$10
Final Result
Remote Client
Database Engine Site

(b)
Stored procedure. The client invokes the stored procedure already stored in the server. The server then executes all the SQL queries in the procedure and sends only the final result back.

Figure 2.1: Interactive query vs Stored Procedure

the client and server regardless of the number of SQL Queries in the procedure. In the example in figure 2.1, the stored procedure reduces the over-the-network round trip needed from 2 to 1. This reduction in server-client communication translates to reduced procedure latency and improved performance overall. This improvement is more pronounced for more complex queries.

2.7 Java Native Interface (JNI)

VoltDB is written in Java but DPDK and F-Stack both are in C. So we need some bridge to connect the socket logic written in C to VoltDB's codebase in Java. I do this using Java Native Interface (JNI). JNI allows a Java program to call native C/C++ methods as if it were a Java method. This native method invoked runs within the JVM and has access to the Java environment and the Java object instance (or class if static). Additionally, JNI also allows the C code to access the Java instance's fields and invoke its methods.

Chapter 3

Previous works

Since the advent of databases, the database community has primarily focused on reducing query execution time to improve database performance.[1][2]. Common approaches include generating better query plans [3] such as adaptive cardinality estimation and planning[18][19], or executing plans faster with faster algorithms[4], application-specific storage formats[5], and in-memory data location [6][7]. Additionally, there has been work in characterizing and optimizing other latency-critical components in databases like buffer management, locking, latching, and scheduling.[8]

Several works that have acknowledged significant overhead from over-the-network communication i.e. category (iii). Hu et. al. observed that more than 50% of the client-side end-to-end round trip time is spent in network communication between the server and the database.[20] They also reported a 59% decrease in end-to-end latency when switching from interactive query to stored procedure. This is why most major databases provide some form of stored procedure feature.[21][22][23][7][24].

Recent works have also focused on reducing network stack overhead i.e. category (ii). Raasveldt and Muhleisen rethink client-server communication protocol by focusing on faster data serialization.[25] Kernel bypassing is also gaining popularity in the industry. Yellowbrick and ScyllaDB, for example, use DPDK to bypass the kernel for networking.[26][27]

Yellowbrick is a SQL database for data warehouse applications spanning multiple physical nodes. Its OLAP workloads workload involves shuffling large volumes of data. Yellowbrick uses kernel bypass (with an additional option to use RDMA) in communication between the nodes to get up to 70% speedup in query execution.[26] Similarly, ScyllaDB employs an open-source framework Seastar to bypass the kernel with zero buffer copy to improve the network performance.[27][28]

While network stack optimization work is limited in the database community, it has received significant attention from the operating system and networking community. Over the years, the Linux networking stack has become much faster.[29][30] Despite these efforts, the overhead from locking, packet processing, buffer copy, buffer management, and scheduling makes it unsuitable for ultra-high performant networking to utilize the capacity modern NICs are capable of.[13][31] Increasing number of solutions tend to bypass the kernel networking stack partially or completely. The addition of eXpress Data Path (XDP) to the kernel provides a way to bypass the TCP stack from within the kernel.[32][33] Similarly, Stackmap provides a custom in-kernel path for performant packet processing.[34] These solutions only bypass the kernel partially. Many solutions choose to bypass the kernel completely and run the whole networking stack in userspace. Most of these use DPDK[35]. It provides userspace programs access to raw ethernet packets. Both Yellowbrick and ScyllaDB use DPDK under the hood. Additionally, there are several open source userspace TCP/IP stacks like F-Stack[15] and mTCP[36] utilizing DPDK.

Chapter 4

Microbenchmark: Echo Server

There are two main reasons I worked on a microbenchmark before moving on to an actual database implementation:

1. It provides a lower barrier to implementing, integrating, and modifying features. This meant that I could try different optimization setups to find the best ones to implement in a database.

2. It allows focusing solely on networking. A fully-fledged database has several components like scheduling, queuing, and locking which adds complexity to understanding a breakdown of a request's lifespan.

4.1 Transport Protocol

I experiment with TCP and UDP using **sockperf** to measure the latency and throughput over the network.[37] **sockperf** is an open-source network benchmarking utility by Mellanox. It allows using either UDP or TCP as the transport protocol. It also has the option to run the benchmark on a single client ping-pong mode to see how latency varies with different packet sizes or multi-client load mode to see how latency varies with throughput.

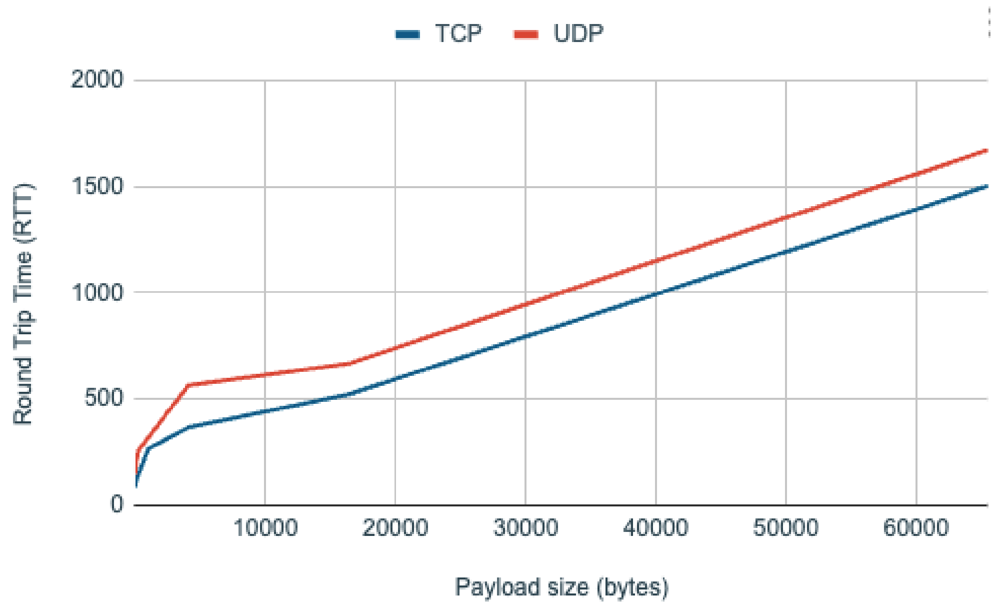


Figure 4.1: RTT vs payload size when using different transport protocols

For this experiment, I run **sockperf** on two physical machines connected over a 1 Gbps connection over the internet. One machine acts as a server and the other as a client.

Figure 4.1 shows the result of the experiment with payload size ranging from 16 bytes to 64KB. TCP consistently has a lower round trip time than UDP despite being the "heavier" of the two in terms of processing overhead. This probably is due to the fact that TCP stack has been highly optimized in Linux with features like Generic Receive Offload (GRO) and TCP Segmentation Offload (TSO).[38] Besides, TCP also employs features like congestion control to optimize traffic.

4.2 Networking Stack

`sockperf` uses the Linux networking stack and does not have immediate support to change the networking stack. So I wrote my own echo server/client programs to measure the latency and throughput. The client is a Java program using the default Linux networking stack and is the same throughout all experiments whereas the server implementation varies in each experiment setup. Regardless of the setup, the server is single-threaded and uses some form of IO multiplexing to handle multiple clients. I use the following 3 different server setups:

1. **Using Java NIO:** This setup uses Java NIO's (New Input/Output) Selector API[1] for multiplexing and SocketChannels[2] for socket operations. Both use Linux's epoll and other socket operations under the hood. It is the base case for comparing other setups

2. **Using JNI and Linux socket** This setup involves the main program in Java interacting with a native C code via JNI. The C code uses Linux socket APIs including multiplexing and socket read and write. Listings 4.1 and 4.2 provide a pseudocode to show how this works. The Java program calls the native C method to listen to client requests. The native method uses `epoll` for IO multiplexing. When there is an incoming client request, the native method invokes a callback to a Java method to process the request. This setup acts as a middle ground between the other two setups. Comparing it with the first setup gives an idea of how efficient (or inefficient) this model of using JNI is. Comparing this setup with the next one using DPDK gives a fairer measure of the improvement solely due to DPDK.

3. **Using JNI and DPDK socket** This setup is similar to the above- the major difference being that it uses DPDK APIs instead of Linux socket APIs for both IO multiplexing and socket operations. This bypasses the kernel to do networking completely in userspace. Instead of using DPDK APIs directly, I use F-Stack[15] which builds on top of DPDK to expose POSIX-like APIs with userspace TCP/IP stack. The design of this setup is also similar to the previous as depicted by pseudocodes in 4.1 and 4.2.

For each setup, I measure the average round trip time by varying i) the message size, and ii) the number of concurrent clients.

Figure 4.2 shows the performance of different server setups for varying payload sizes from 32 bytes to 16 KB. The two servers using Linux sockets have very similar performance. A

[1]https://docs.oracle.com/javase/8/docs/api/java/nio/channels/Selector.html
[2]https://docs.oracle.com/javase/8/docs/api/java/nio/channels/SocketChannel.html

```java
// Java code
class JNIEchoServer {
    load_C_code();
    public native void init(); // call native function in C
    public native void listen_for_event();  // call native function
    public void handleAccept(int fd) {
        // authenticate and process new client connection
    }
    public void indicateReadyForRead(int fd) {
        // read data from fd and process
    }
    public void handleReadyForWrite(int fd) {
        // handle write to fd
    }
}
```

Listing 4.1: Java pseudocode echo server with JNI

```c
// C code
void JNICALL init() {
    // initialize epoll
}
void JNICALL listen_for_event() {
    while (true) {
        // wait for new event {type, fd}
        if (event.type is new connection)
            FSelectJava.handleAccept(event.fd)
        else if (event.type is readReady)
            FSelectJava.indicateReadyForRead(event.fd)
        else if (event.type is writeReady)
            FSelectJava.handleReadyForWrite(event.fd)
    }
}
```

Listing 4.2: C pseudocode of JNI

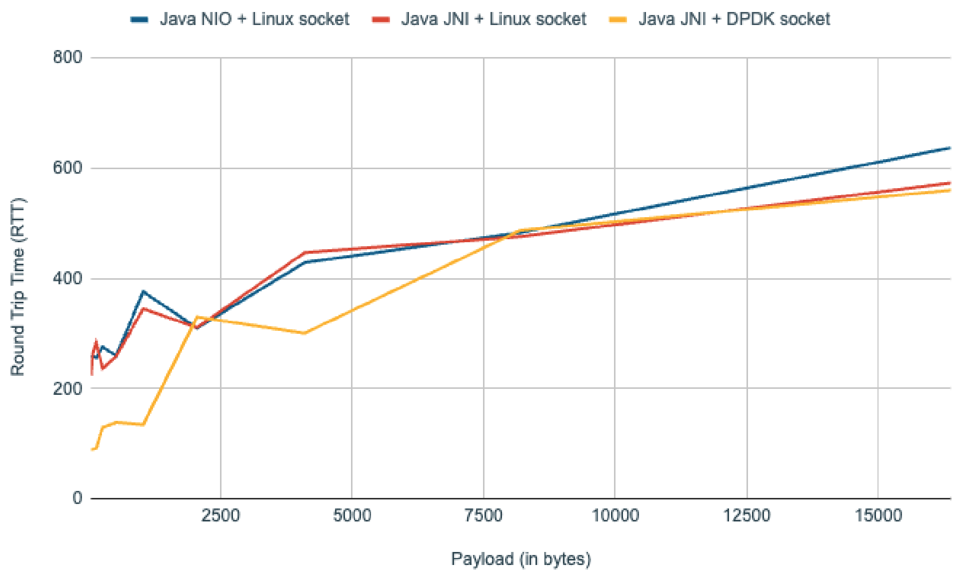


Figure 4.2: RTT vs payload size for one synchronous client

similar story is seen in the RTT vs throughput graph in figure 4.3 as well. The red and blue lines are within a margin of error if we consider the slight variance in network latency.

The server with DPDK sockets performs much better than both the other servers. The performance difference is higher with smaller payload sizes. Particularly for message sizes under the MTU (Maximum Transmission Unit) of the network (1500 bytes), using a DPDK socket reduces the RTT by 45-65% compared to using a Linux socket. This range is perfect for OLTP workloads since these workloads feature short-run/small responses. My experiments show that the result size for the TPCC workload ranges between 34 and 1070 bytes.

The RTT vs throughput graph shows an even bigger gap between the performance of Linux and DPDK sockets. On a single thread, DPDK can sustain almost double the throughput compared to Linux sockets (34 MBps vs 65 MBps).

One reason for DPDK's superior performance is that DPDK runs in polling mode as opposed to the Linux socket which uses interrupts. Polling mode causes the echo server with DPDK to have a high CPU usage even when there is low or no load. For the Linux stack, CPU usage increases proportional to the load. This can be seen in figure 4.4. At high load, however, the Linux stack could use more than 100% CPU on a multi-core machine (100% =

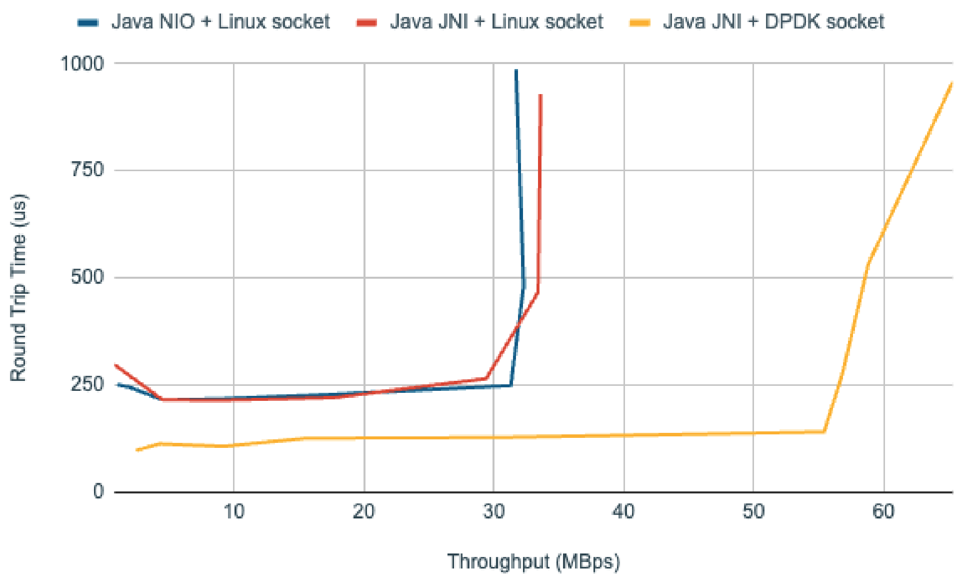


Figure 4.3: RTT vs throughput for 256 bytes message. Throughput is varied by changing the number of clients

1 CPU). This is because Linux handles interrupts raised by NIC in a separate background thread. So, even a single-threaded echo server with Linux stack, in reality, uses 2 threads for packet processing.

The following lessons can be learned from the above echo server microbenchmark:

- TCP performs better than UDP in Linux
- Switching from Java's NIO API to JNI for socket operations does not affect the performance of the server
- For packet size below the network's MTU, DPDK outperforms the Linux stack. It decreases the latency by more than 45% latency and achieves almost double the throughput.
- At high workload, Linux could use more CPU for networking than DPDK.

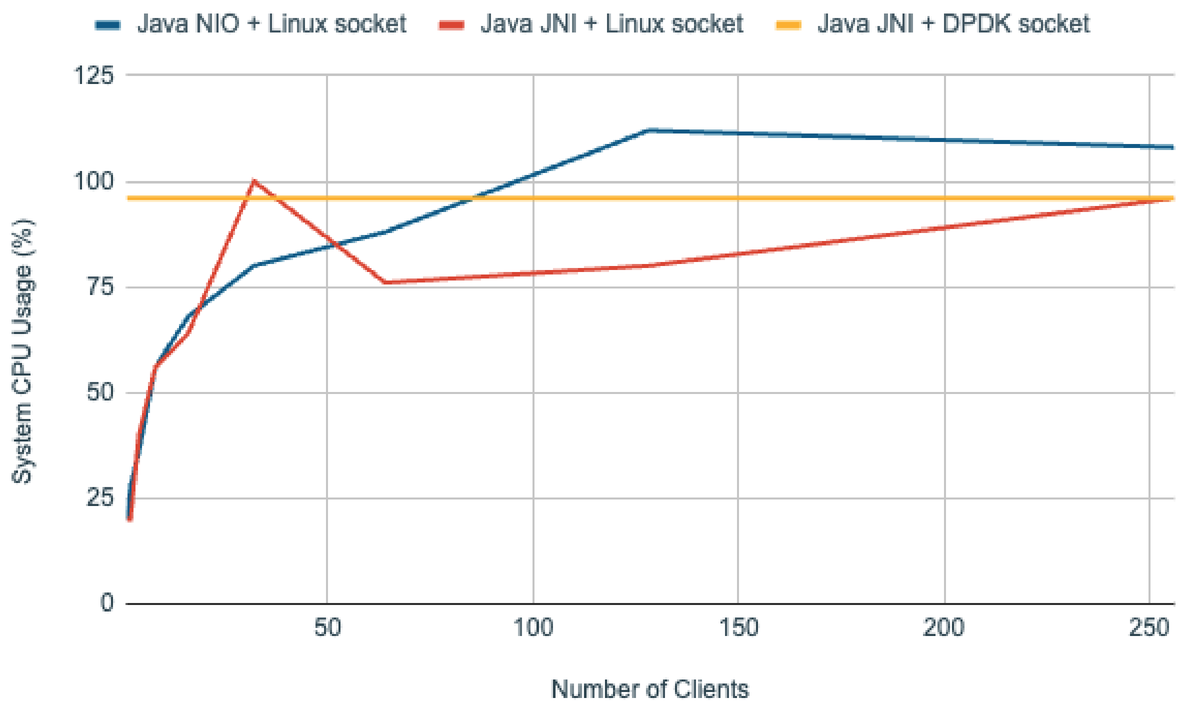


Figure 4.4: CPU usage for different echo server setups when varying the number of clients. 100% CPU usage is equivalent to 1 CPU being fully used. The server has 40 CPUs

Chapter 5

VoltDB

I use VoltDB[7] as the database for my work. VoltDB is a high-performance in-memory OLTP relational database. It is a full-fledged relational database and has a community version available that runs on commodity hardware.[39] It is primarily written in Java but internal data management and query execution is done in C++. It allows users to write Stored Procedures in Java.

5.1 Architecture

VoltDB uses a shared-nothing architecture based on H-Store.[40] Data is partitioned horizontally across execution sites running on different cores. This allows the database to scale with the number of cores. Figure 5.1 shows the architecture of VoltDB.

In addition to one thread per execution site, VoltDB allows running multiple network threads. The network threads and execution site threads communicate asynchronously via queues. In addition, there is a Client Acceptor thread responsible for accepting and authenticating new client connections. All network operations use Java's NIO (New I/O) APIs which use the Linux network stack under the hood.

When a new client connection request arrives at the server, the Client Acceptor thread accepts and authenticates it. The Client Acceptor then registers the client with one of the

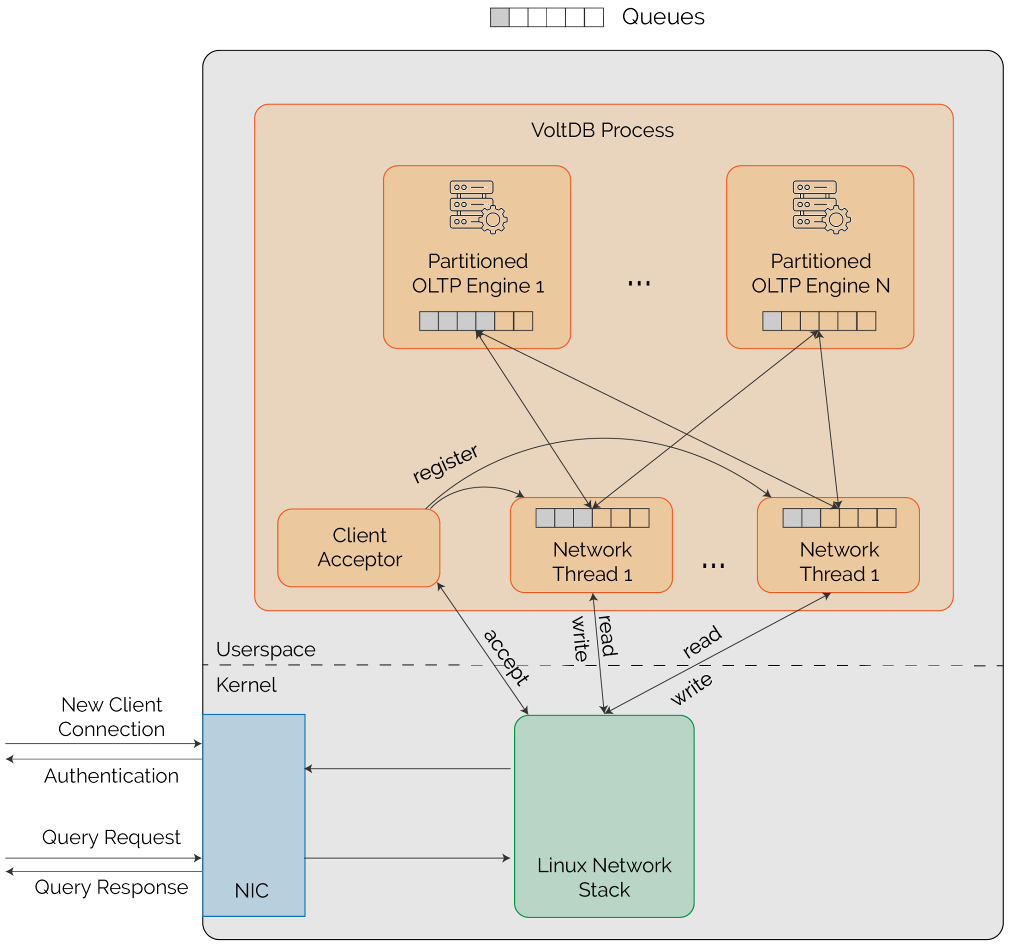


Figure 5.1: VoltDB architecture with a focus on networking

network threads chosen in a round-robin fashion. This network thread is responsible for further communication with the client. When the client sends a request, the network thread reads the request, determines the partition responsible for handling the request, and passes it to the partition's queue. The partition then picks up the request from the queue, executes the query, and writes the response back to the network thread's queue. The network thread then picks up the response and sends it back to the client.

5.2 Optimization

The echo server microbenchmark provided two key insights: kernel bypassing improves the networking performance and TCP performs better than UDP in practice. So I abandon the

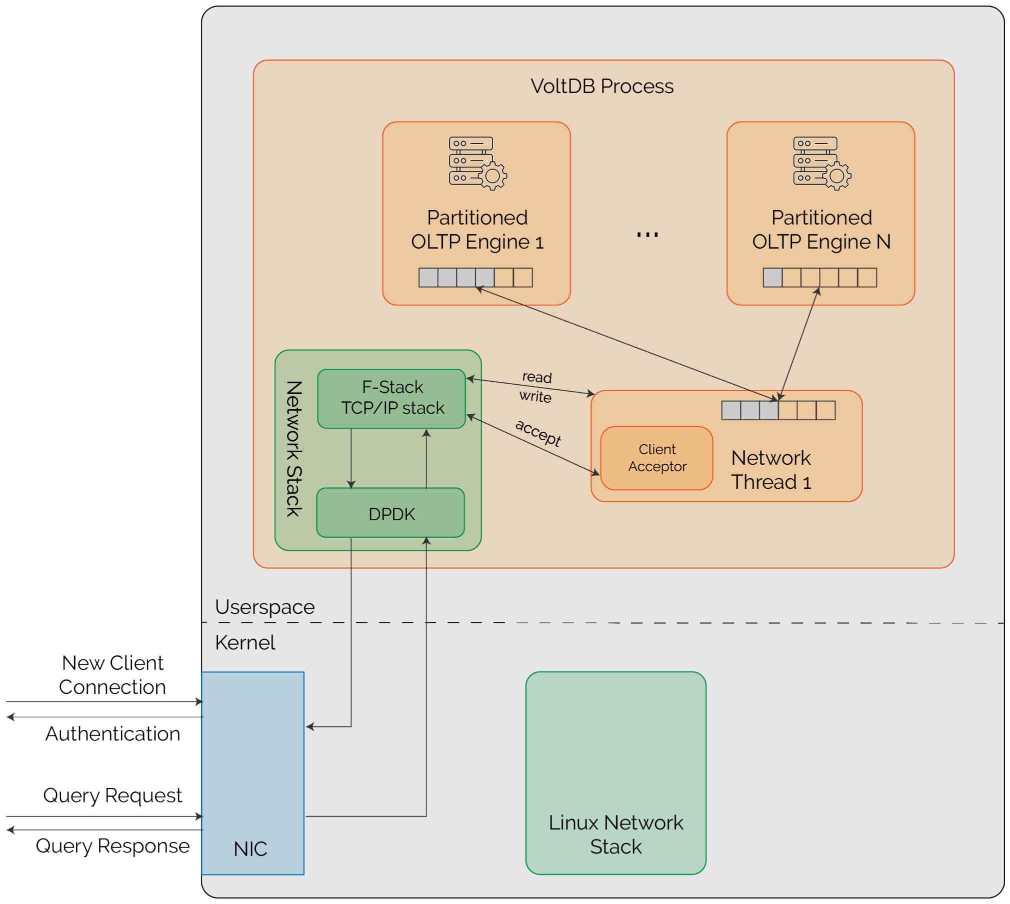


Figure 5.2: VoltDB architecture modified to bypass kernel using DPDK and F-Stack. Since F-Stack is single-threaded, all network activities are consolidated in a single networking thread.

prospect of changing the transport protocol. Instead, I implemented a networking stack for VoltDB that bypasses the kernel using DPDK and F-Stack.

5.2.1 Kernel bypass

The choice of DPDK and F-Stack requires us to write the core networking logic in C/C++. But VoltDB is written in Java. So similar to the echo server, I use JNI to bridge the Java to C communication. VoltDB's codebase is modular which makes it possible to rewrite only the core networking module with minimal changes to other parts of the codebase. The new architecture is shown in figure 5.2. There is no change in how the execution sites work or the communication between the execution sites and the network thread. The network model, however, is a bit different:

- No syscall is required for socket operations since everything runs in userspace. Instead, socket operations turn into native method invocation via JNI.

- Number of network threads is limited to 1. This is because F-stack is single-threaded and needs a dedicated NIC. In fact, one of the reasons F-stack is able to achieve high performance is because it specializes in single-thread networking by getting rid of any locking.

- Client Acceptor is moved within the network thread. In the original VoltDB architecture in 5.1, the Client Acceptor thread is separate from the networking threads. This design prevented any effect of backpressure handling during high workload to new client connection requests. But the single-threaded nature of F-stack does not allow this. So I pass the method to accept the client connection as a callback to the network thread. The network thread listens for new clients on the socket and invokes this callback to authenticate the client.

- Network packets are received via polling instead of interrupt. In Linux, the arrival of data in the NIC is notified by raising a hardware interrupt. A background thread handles this interrupt to read the data from the NIC. Since DPDK runs completely in userspace, it cannot listen to hardware interrupts. Instead, it polls the NIC registers periodically to check if there is any new data.

Chapter 6

Experiments

The goal of the experiments is twofold. The first is to quantify the contributions of different components of the database server to the latency numbers observed by the client. The second is to compare the performance of the database at different optimization levels and understand the source of the difference in performance.

6.1 Infrastructure

Two on-premises machines are used for the experiments. The server machine has x86_64 architecture with 40 CPUs and 4 10Gbps Intel X710 network interfaces running Ubuntu 22.04.4 LTS with 5.15.0-105-generic Linux kernel. The client machine also has x86_64 architecture but with 32 CPUs and 1 1Gbps Intel I225-V network interface running Ubuntu 22.04.1 LTS with 6.2.0-36-generic Linux kernel.

6.2 Benchmarks

I use TPCC[41] and Retwis[42] benchmarks for the experiments.

Table 6.1: Mix of Transactions in TPCC Workload

Transaction	Number of queries	Proportion in workload
New Order	8-18	45%
Payment	5	43%
Delivery	4	4%
Stock Level	2	4%
Order Status	3	4%

6.2.1 TPCC

TPCC is an industry-standard benchmark for OLTP databases. From TPCC's own documentation, "TPCC involves a mix of five concurrent transactions of different types and complexity either executed online or queued for deferred execution. The database is comprised of nine types of tables with a wide range of record and population sizes. ... While the benchmark portrays the activity of a wholesale supplier, TPC-C is not limited to the activity of any particular business segment, but, rather represents any industry that must manage, sell, or distribute a product or service."

The 5 transactions and their mix in the TPCC workload variant I use are shown in table 6.1. These are stored as procedures in VoltDB and the client (also known as terminal in TPCC) invokes them synchronously. The performance of TPCC workload is measured in terms of tpmC defined as the number of new order transactions per minute. tpmC gives a more realistic metric to measure the "business throughput" since a new order is the only transaction that generates revenue.

6.2.2 Retwis

Retwis is a Twitter clone simulating an online social media platform. The application allows users to create posts, follow each other, see a timeline of the most recent posts of people they follow, and view specific people's posts. It was introduced by Redis[42]. It is much simpler than TPCC and has only 1 query per transaction. I run a workload with 100% Get Posts transaction which fetches up to 30 most recent posts from a user. So this benchmark only involves retrieving data from the database.

6.3 Understanding overhead from different database components

Understanding the overhead from different database components is key to optimizing the overall performance of the database. I take two different approaches to breaking down the overhead during database runtime:

6.3.1 Profile latency of each request for a single synchronous user

While measuring CPU overhead is helpful, it does not capture some sources of latencies like scheduling or when the CPU is idle waiting for something. So I measure a breakdown of the time spent in each different component in the latency critical path. For this experiment, I only run 1 synchronous client to track the lifespan of the client's requests from the time it arrives at the server's NIC to the time the response is written to the NIC to send back to the client.

To measure time spent in different components, I add tracepoints to different functions on the latency-critical path. Table 6.2 shows a list of all the tracepoints instrumented for measurement along with the thread the tracepoint runs in. The tracepoints starting with `probe` are in kernel. These are instrumented using `perf`. Similarly, those starting `voltdb` are within VoltDB code, those with `fstack` are in F-stack, and those with `dpdk` are in DPDK core. These are in userspace and are instrumented using a custom code. After recording these tracepoint events, I group the tracepoints into different functional groups as shown in table 6.3. This grouping helps analyze the overhead of different functional components in the pathway.

Table 6.2: Tracepoints instrumented to analyze VoltDB

tracepoint	Thread	Description
`probe:net_rx_action`	swapper	Handles RX interrupt raised by NIC
`dpdk:nic_read_ready`	Volt Network	Indicates NIC has packets ready for read
`probe:ip_list_rcv`	swapper	Processes raw packet to a TCP/IP packet
`fstack:ip_input`	Volt Network	Processes raw packet to a TCP/IP packet
`probe:ep_poll_callback`	swapper	Wakes up the network thread sleeping on `epoll_wait`
`probe:epoll_wait__return`	Volt Network	Epoll wait returns
`fstack:epoll_wait__return`	Volt Network	Epoll wait returns
`probe:ksys_read`	Volt Network	Reading incoming data
`probe:ksys_read__return`	Volt Network	Finished reading incoming data
`voltdb:txnqueue`	Volt Network	Queue transaction for processing
`voltdb:txnrecv`	SP Worker	Worker thread dequeued queued transaction
`voltdb:txnsend`	SP Worker	Worker thread queues response for sending
`voltdb:responsequeue`	Volt Network	Networking thread received response to send
`probe:ksys_write`	Volt Network	Process response bytes to send back to the client
`probe:dev_queue_xmit`	Volt Network	Write response bytes has been written to the NIC
`probe:dev_queue_xmit__return`	Volt Network	Response bytes has been written to the NIC
`dpdk:nic_write_return`	Volt Network	Response bytes has been written to the NIC

Table 6.3: Grouping tracepoints for breakdown for VoltDB using Linux network stack

Component	**Start**	**End**
RX interrupt handling	`probe:net_rx_action`	`probe:ip_list_rcv`
RX TCP/IP Handling	`probe:ip_list_rcv`	`probe:ep_poll_callback`
Epoll wait	`probe:ep_poll_callback`	`probe:epoll_wait__return`
RX Read	`probe:epoll_wait__return`	`probe:ksys_read__return`
Network-Worker thread scheduling	`probe:ksys_read__return`	`voltdb:txnrecv`
Transaction Work	`voltdb:txnrecv`	`voltdb:txnsend`
Worker-Network thread scheduling	`voltdb:txnsend`	`voltdb:responsequeue`
TX TCP/IP processing	`voltdb:responsequeue`	`probe:dev_queue_xmit`
TX Write to NIC	`probe:dev_queue_xmit`	`probe:dev_queue_xmit__return`

Chapter 7

Results

7.1 Latency-throughput

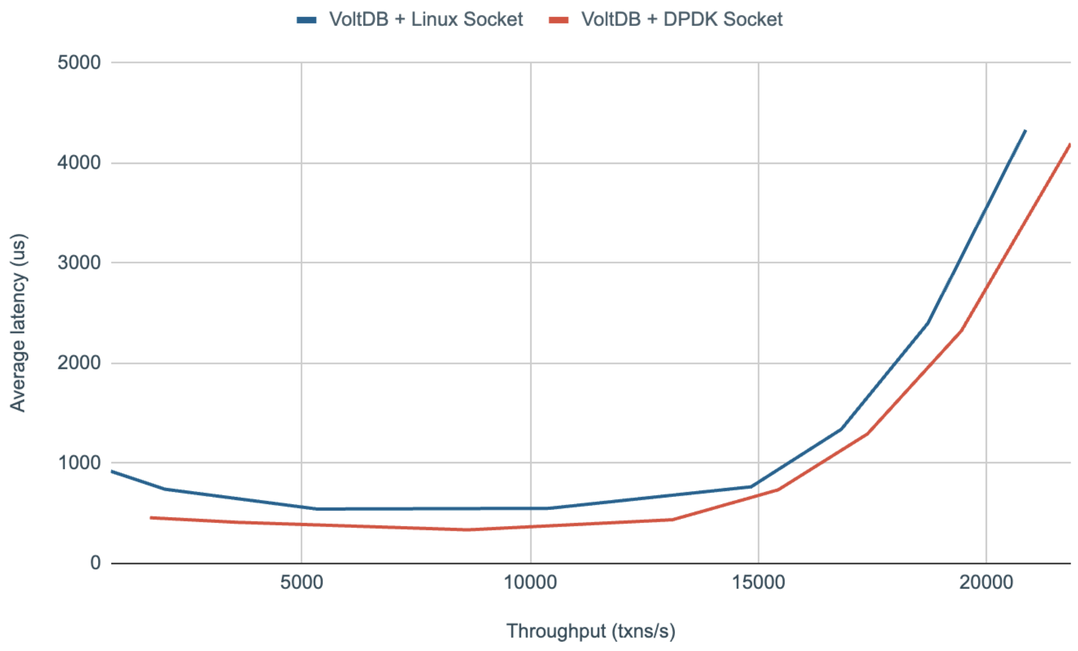


Figure 7.1: Latency vs throughput for TPCC workload on VoltDB with two different network stacks. Throughput is varied by varying the number of clients.

Figure 7.1 shows the performance of VoltDB running a single execution site thread for the TPCC workload. DPDK network stack outperforms Linux stack for all workload levels. For

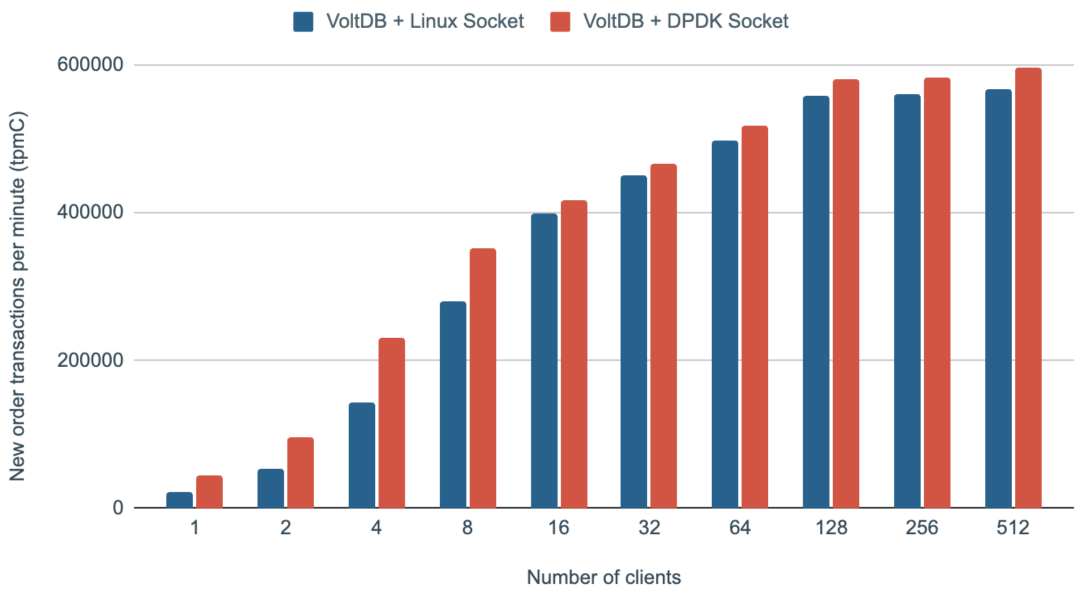


Figure 7.2: Number of new order transactions per minute (tpmC) for different number of terminals (clients)

the same throughput, VoltDB with DPDK stack is able to achieve 10-30% lower latency. The gap is lower for moderate load of 4,000-15,000 transactions per second, but gets higher for low or high workload. This is probably because Linux batches concurrent reads and writes over the socket. This batching helps lower the throughput when increasing the number of clients starting from 1. But when the concurrency is high, the overhead from other parts (like locking and packet processing) dominates the improvement from batching.

Similarly, the DPDK stack outperforms the Linux stack in the number of new order transactions per minute (tpmC) metric as well. For a single client, the DPDK stack helps VoltDB achieve 100% more new order transactions per minute compared to Linux stack. At a higher workload, the advantage of using DPDK drops and stabilizes to around 4% improvement over Linux stack.

A similar result can be seen for Retwis GetPosts transaction as well in figure 7.3. DPDK network stack outperforms Linux stack by up to 40%. This improvement is better than what we saw in TPCC workload. This is because retwis transactions are simpler than TPCC and thus larger proportion of latency is spent in networking.

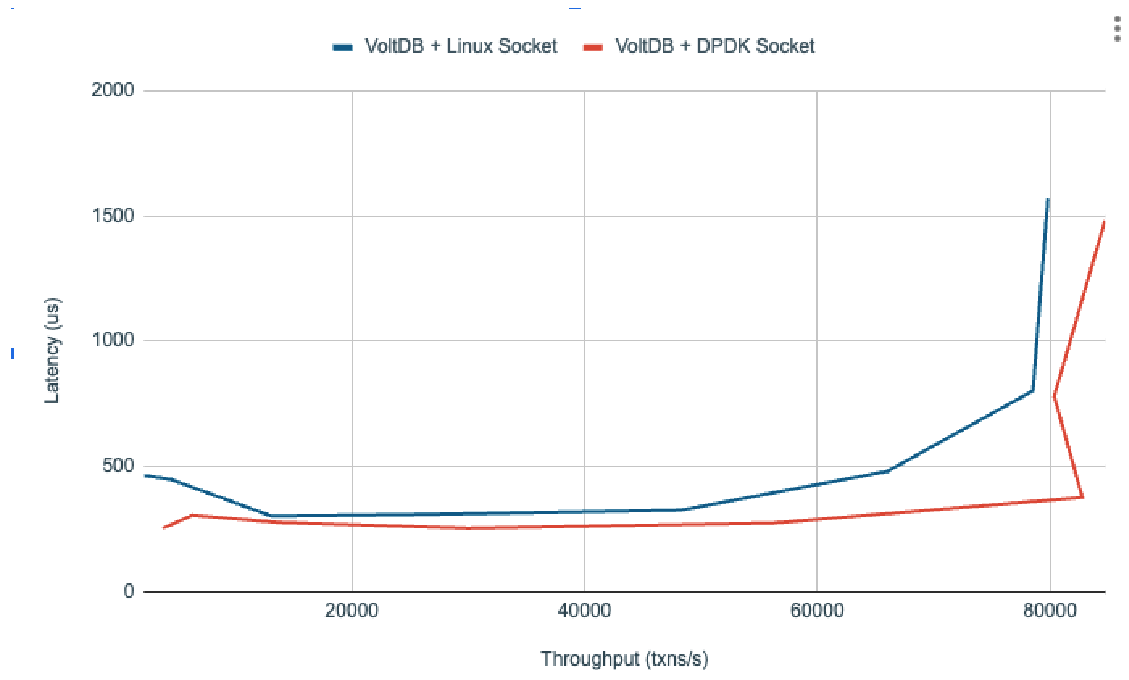


Figure 7.3: Latency vs throughput for Retwis running 100% GetPosts workload on VoltDB with two different networking stacks. Throughput is varied by varying the number of clients.

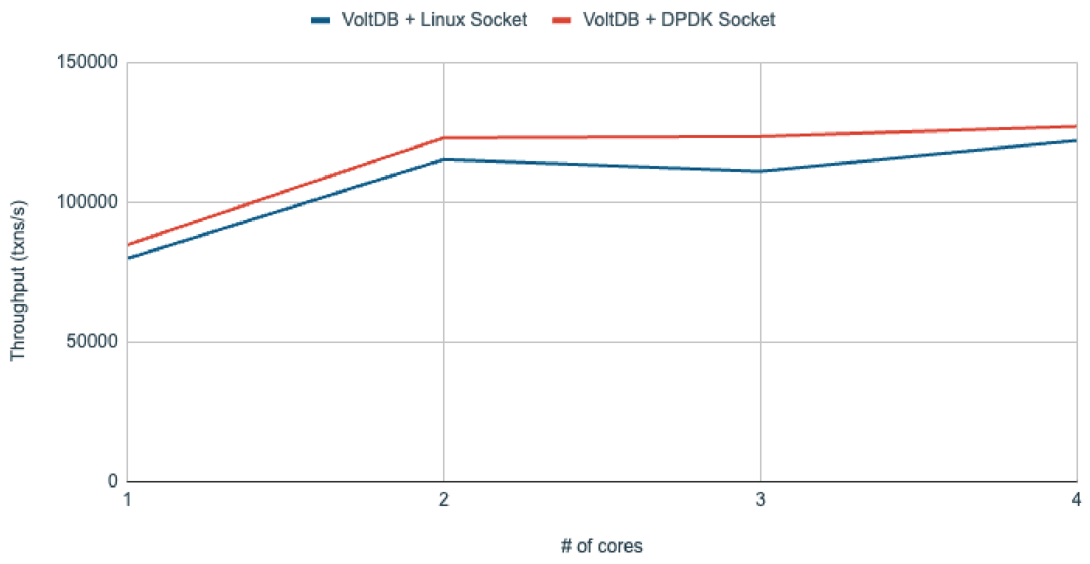


Figure 7.4: Throughput vs the number of execution cores for Retwis GetPosts workload on VoltDB.

Both the versions saturate at around the same throughput of 80K transactions per second. The execution site thread maxes its capacity at this point. This can be inferred from figure 7.4 where we can get more throughput by increasing the number of execution threads to 2. However, increasing the number of cores beyond that does not increase the throughput of the system. At this point, the network is saturated and the system is network-bound.

7.2 Breaking down the server time

The performance boost mentioned in the previous section comes solely from the change in the networking stack. This can be verified by analyzing the lifespan of a single client request. I use Retwis's GetPost transaction for this. GetPost is a simple get query that makes it easier to trace.

Figure 7.5 shows a distribution of time spent in different components of the database. Table 7.1 shows a finer time breakdown of a single client request as it goes through the server. The original VoltDB, a GetPost request spends around 15% of the total end-to-end latency in server networking stack. Most of this overhead comes from blocking IO Multiplexing (49%) and TCP/IP packet processing (36%).

Switching out the Linux socket with the DPDK socket reduces the time a request spends in the server by around 38% (178us vs 111us). This means DPDK sockets enable VoltDB to process requests faster. This performance boost can be mainly attributed to faster networking. The DPDK stack spends 80% (44us) less time in networking compared to the Linux stack. This can be attributed to the following overheads in Linux which are not present in DPDK

- Context switches due to syscalls when calling `read`, `write`, or `epoll_wait`
- RX interrupt handling (DPDK uses polling instead of interrupts)
- Blocking `epoll_wait` (everything is non-blocking in DPDK)

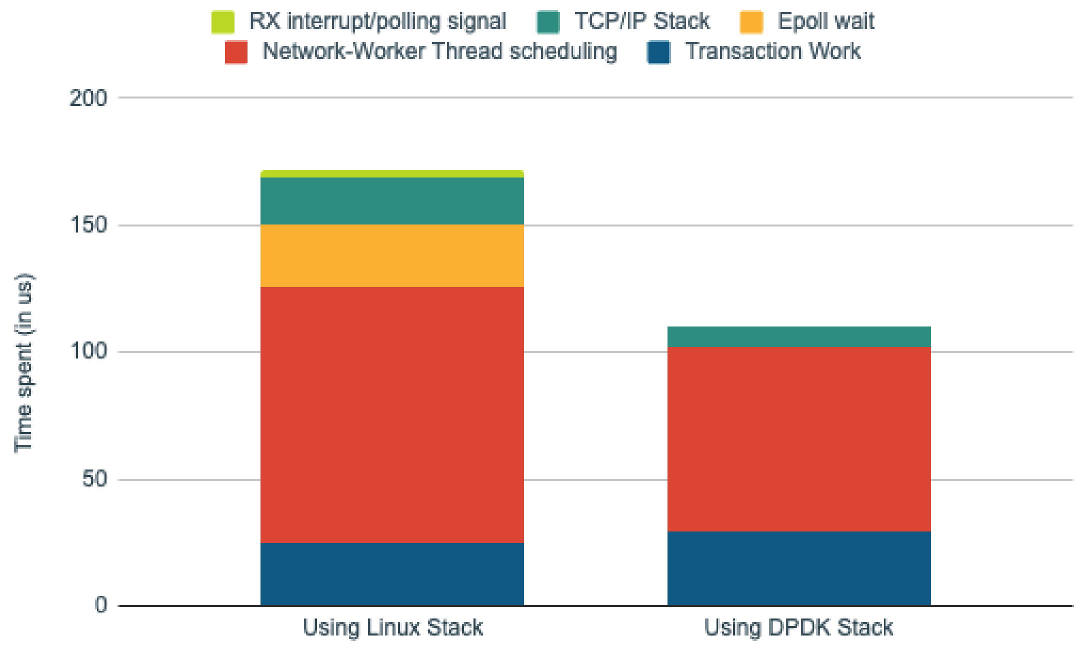


Figure 7.5: Breakdown of time spent in server for each GetPost request Retwis

- Locking to access NIC and TCP queue

Th performance boost in networking translates to a 44% increase in the throughput of the server.

Besides networking, VoltDB also spends a significant amount of time (56%) in thread scheduling between the network thread and the worker thread. Accounting for all of this, the actual transaction work is only 14% of total server time.

Component	Linux networking stack (us)	DPDK networking stack (us)
RX interrupt handling	3.11	0.31
RX TCP/IP Handling	4.1	1.1
Epoll wait	25.5	1.3
RX Read	6.49	0.31
Network-Worker thread scheduling	49.1	42
Transaction Work	24.7	29.1
Worker-Network thread scheduling	52	30.7
TX TCP/IP processing	5.7	2.3
TX Write to NIC	7.2	3.4
Total time in server	178.03	110.58
End-to-end latency	350	250

Table 7.1: Breakdown of server time for an average GetPost request in Retwis benchmark. A description of the components can be found in table 6.3

Chapter 8

Conclusion

In this thesis, I dissect the overhead associated with different components of a high-performance modern database, focusing particularly on the networking stack. I find a significant amount of time and CPU cycles is spent in the operating system networking stack. Besides useful work to read and write data, this includes overhead from in-kernel packet processing with locking, blocking IO multiplexing, and NIC interrupt handling to name some. These bottlenecks are rooted in the Linux networking stack, which is designed for generic applications rather than high-performance networking.

To tackle this network overhead, I optimized the network module of the database to use DPDK to bypass the kernel network stack and run networking completely in userspace. This helped achieve up to 80% faster networking and up to 40% lower end-to-end latency. Along with the decrease in latency for each client request, the throughput also increased by up to 10%. This improvement comes at the cost of dedicating a whole NIC to an application and high CPU usage regardless of workload. Both of these factors should not be an issue in today's time when having multiple CPUs and NICs in servers is common.

8.1 Implications and Future Work

8.1.1 Extension to clients

This work focused only on the server networking stack. This is because kernel bypass requires administrator (`sudo`) access and specialized hardware (CPU, NIC, crypto engine, etc)[43]. However, one cannot guarantee such hardware in a client system. So replacing the client networking stack with DPDK is not practical. However, more and more modern hardware is designed with this flexibility in mind. This could make the necessary hardware more commonplace in the future and possibly bypass the kernel in the client too. Another possibility to achieve a similar outcome could be to use tools like XDP which lets one bypass most of the kernel stack from within kernel.[32]

8.1.2 Open-source Instrumentation tool

The scripts for measuring the breakdown of server time are open source at https://github.com/kafleprabhakar/latencyBench. It contains the code for adding and recording tracepoints for both kernel and userspace functions. It also contains a Python script you can configure to analyze the recorded events to create a breakdown of an application run. In addition, the repository also has an echo server example which has been instrumented to give a breakdown of the lifetime of a request in server. This can be used as a low-overhead tool to analyze the performance bottlenecks for any system.

8.1.3 Multi-node databases

The focus of this work was database settings with only one logical (and physical) node. This meant that the query execution part did not involve any data shuffle or message passing that might require communication between isolated nodes. This communication is often done over the network and could benefit highly from kernel bypassing. Yellowbrick already takes advantage of kernel bypass for a similar purpose in OLAP workloads.[26]

8.1.4 Use of Smart NICs

This work explored offloading networking from kernel to userspace. This could be taken a step further with Smart NICs that allow offloading packet processing to the NIC. Recent works have explored the possibility of programming the whole TCP/IP stack to a Smart NIC.[31][44] This could reduce the networking overhead and free up computing capacity to do more useful work, thus increasing the throughput of the database. This also decouples the packet processing site (the NIC) from the operating system stack. This decoupling could make it easier to scale networking capacity independent of the OS stack.

Appendix A

Linux Networking Stack

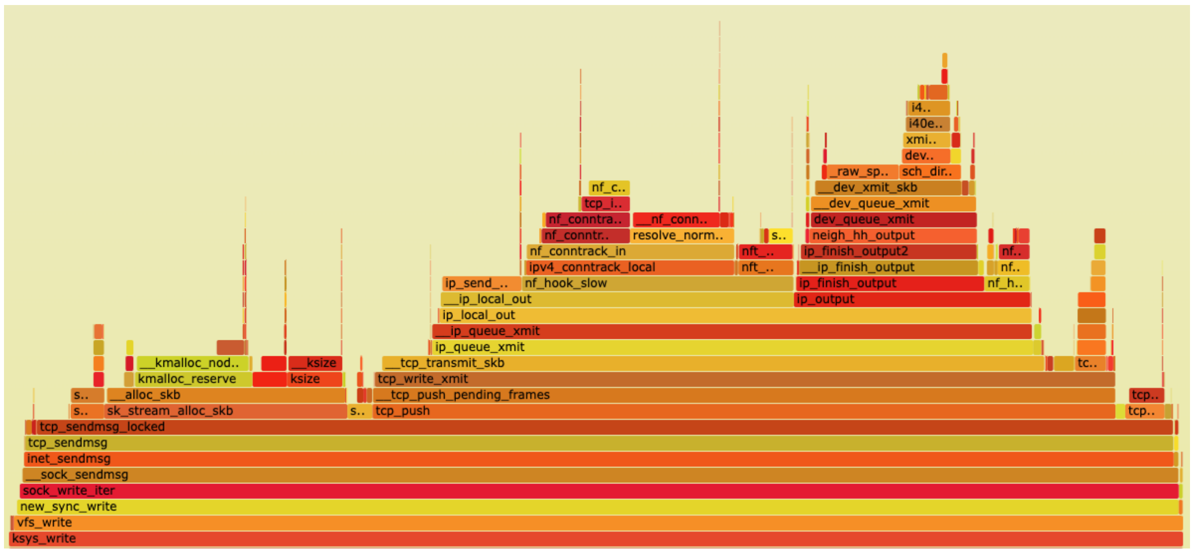
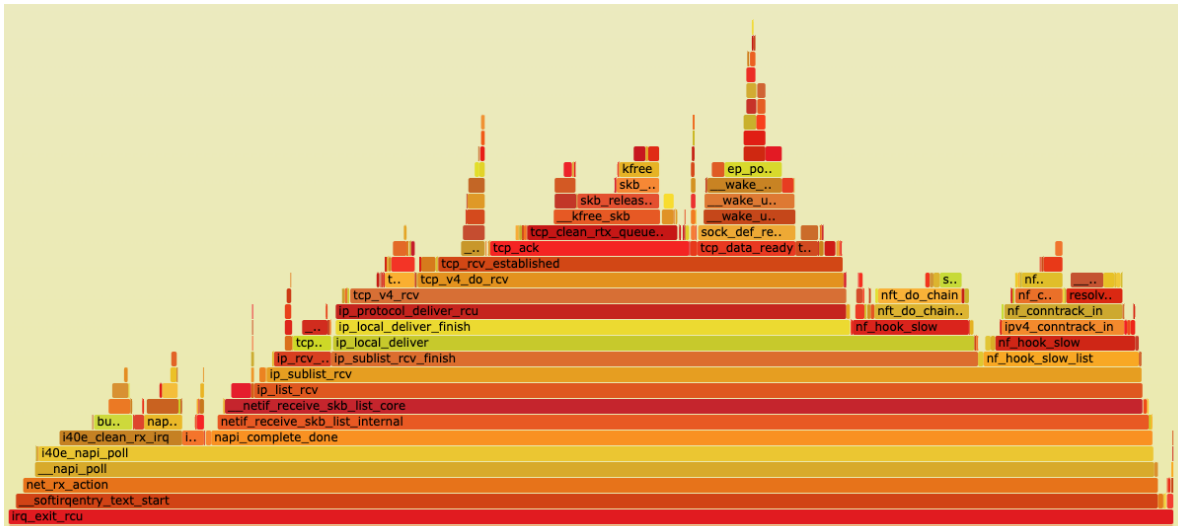
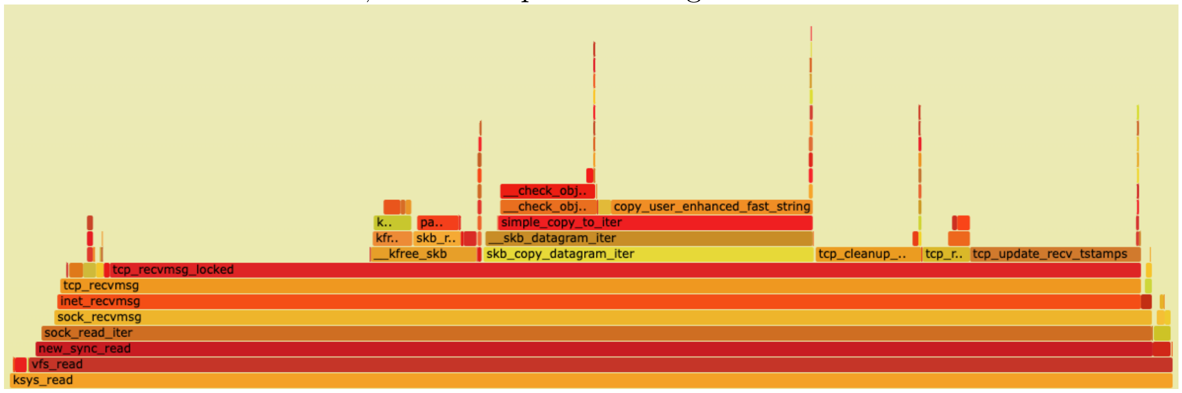


Figure A.1: Flamegraph of server writing 256 bytes of data to the NIC for transmission



(a)

When data arrives, NIC interrupt is handled by calling `net_rx_action` in `swapper` thread. At the end, it wakes up the waiting read thread shown in A.2b



(b)

Reading data from TCP buffer after being woken up by swapper thread

Figure A.2: Flamegraph of server receiving 256 bytes of data from NIC

References

[1] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '98, Seattle, Washington, USA: Association for Computing Machinery, 1998, pp. 34–43, ISBN: 0897919963. DOI: 10.1145/275487.275492. URL: https://doi.org/10.1145/275487.275492.

[2] E. Wong and K. Youssefi, "Decomposition—a strategy for query processing," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 223–241, Sep. 1976, ISSN: 0362-5915. DOI: 10.1145/320473.320479. URL: https://doi.org/10.1145/320473.320479.

[3] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "Query optimization through the looking glass, and what we found running the join order benchmark," *The VLDB Journal*, vol. 27, no. 5, pp. 643–668, Oct. 2018, ISSN: 1066-8888. DOI: 10.1007/s00778-017-0480-7. URL: https://doi.org/10.1007/s00778-017-0480-7.

[4] Y. Zhao, P. M. Deshpande, and J. F. Naughton, "An array-based algorithm for simultaneous multidimensional aggregates," *SIGMOD Rec.*, vol. 26, no. 2, pp. 159–170, Jun. 1997, ISSN: 0163-5808. DOI: 10.1145/253262.253288. URL: https://doi.org/10.1145/253262.253288.

[5] M. Stonebraker, D. J. Abadi, A. Batkin, *et al.*, "C-store: A column-oriented dbms," in *Making Databases Work: The Pragmatic Wisdom of Michael Stonebraker*.

Association for Computing Machinery and Morgan & Claypool, 2018, pp. 491–518, ISBN: 9781947487192. URL: https://doi.org/10.1145/3226595.3226638.

[6] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, New York, USA: Association for Computing Machinery, 2013, pp. 1243–1254, ISBN: 9781450320375. DOI: 10.1145/2463676.2463710. URL: https://doi.org/10.1145/2463676.2463710.

[7] Apr. 2024. URL: https://www.voltactivedata.com/.

[8] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "Oltp through the looking glass, and what we found there," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, pp. 981–992, ISBN: 9781605581026. DOI: 10.1145/1376616.1376713. URL: https://doi.org/10.1145/1376616.1376713.

[9] D. Both and D. Both, "Everything is a file," *Using and Administering Linux: Volume 2: Zero to SysAdmin: Advanced Topics*, pp. 43–66, 2020.

[10] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson, "Tas: Tcp acceleration as an os service," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19, Dresden, Germany: Association for Computing Machinery, 2019, ISBN: 9781450362818. DOI: 10.1145/3302424.3303985. URL: https://doi.org/10.1145/3302424.3303985.

[11] Wikipedia, *Transmission Control Protocol — Wikipedia, the free encyclopedia*, http://en.wikipedia.org/w/index.php?title=Transmission%20Control%20Protocol&oldid=1222972304, [Online; accessed 14-May-2024], 2024.

[12] Wikipedia, *User Datagram Protocol — Wikipedia, the free encyclopedia*, http://en.wikipedia.org/w/index.php?title=User%20Datagram%20Protocol&oldid=1219304722, [Online; accessed 14-May-2024], 2024.

[13] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 65–77, ISBN: 9781450383837. DOI: 10.1145/3452296.3472888. URL: https://doi.org/10.1145/3452296.3472888.

[14] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984, ISSN: 0734-2071. DOI: 10.1145/357401.357402. URL: https://doi.org/10.1145/357401.357402.

[15] Apr. 2024. URL: https://www.f-stack.org/.

[16] May 2024. URL: https://github.com/F-Stack/f-stack/blob/dev/README.md#nginx-testing-result.

[17] S. Eranian, E. Gouriou, T. Moseley, and W. de Bruijn. "Tutorial- perf wiki." (Aug. 2023), URL: https://perf.wiki.kernel.org/index.php/Tutorial (visited on 05/11/2024).

[18] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '00, Dallas, Texas, USA: Association for Computing Machinery, 2000, pp. 261–272, ISBN: 1581132174. DOI: 10.1145/342009.335420. URL: https://doi.org/10.1145/342009.335420.

[19] A. Deshpande, Z. Ives, V. Raman, *et al.*, "Adaptive query processing," *Foundations and Trends® in Databases*, vol. 1, no. 1, pp. 1–140, 2007.

[20] G. Hu, Z. Wang, C. Tang, J. Shen, Z. Dong, S. Yao, and H. Chen, "Webridge: Synthesizing stored procedures for large-scale real-world web applications," *Proc.*

ACM Manag. Data, vol. 2, no. 1, Mar. 2024. DOI: 10.1145/3639319. URL: https://doi.org/10.1145/3639319.

[21] Apr. 2024. URL: https://www.postgresql.org/docs/current/xproc.html.

[22] Apr. 2024. URL: https://docs.oracle.com/database/121/LNPLS/create_procedure.htm.

[23] Apr. 2024. URL: https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html.

[24] A. Eisenberg, "New standard for stored procedures in sql," *ACM SIGMOD Record*, vol. 25, no. 4, pp. 81–88, 1996.

[25] M. Raasveldt and H. Mühleisen, "Don't hold my data hostage: A case for client protocol redesign," *Proc. VLDB Endow.*, vol. 10, no. 10, pp. 1022–1033, Jun. 2017, ISSN: 2150-8097. DOI: 10.14778/3115404.3115408. URL: https://doi.org/10.14778/3115404.3115408.

[26] M. Cusack, J. Adamson, M. Brinicombe, N. Carson, T. Kejser, J. Peterson, A. Vasudev, K. Westerfeld, and R. Wipfel, "Yellowbrick: An elastic data warehouse on kubernetes,"

[27] Apr. 2024. URL: https://www.scylladb.com/2024/01/03/reduce-database-costs/.

[28] Apr. 2024. URL: https://seastar.io/.

[29] Apr. 2024. URL: https://docs.kernel.org/networking/scaling.html.

[30] J. Corbet, "Batch processing of network packets," URL: https://lwn.net/Articles/763056/.

[31] T. Kim, D. M. Ng, J. Gong, Y. Kwon, M. Yu, and K. Park, "Rearchitecting the TCP stack for I/O-Offloaded content delivery," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA: USENIX Association, Apr. 2023, pp. 275–292, ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/kim-taehyun.

[32] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18, Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66, ISBN: 9781450360807. DOI: 10.1145/3281411.3281443. URL: https://doi.org/10.1145/3281411.3281443.

[33] Apr. 2024. URL: https://www.kernel.org/doc/html/next/networking/af_xdp.html.

[34] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-Latency networking with the OS stack and dedicated NICs," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO: USENIX Association, Jun. 2016, pp. 43–56, ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata.

[35] Apr. 2024. URL: https://www.dpdk.org/.

[36] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, Seattle, WA: USENIX Association, Apr. 2014, pp. 489–502, ISBN: 978-1-931971-09-6. URL: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[37] May 2024. URL: https://github.com/Mellanox/sockperf.

[38] "Segmentation offloads," The Linux Kernel Documentation. (), URL: https://www.kernel.org/doc/html/next/networking/segmentation-offloads.html.

[39] "Voltdb," Gitbub. (), URL: https://github.com/VoltDB/voltdb (visited on 05/04/2024).

[40] R. Kallman, H. Kimura, J. Natkins, *et al.*, "H-store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008, ISSN: 2150-8097. DOI: 10.14778/1454159.1454211. URL: https://doi.org/10.14778/1454159.1454211.

[41] "Tpcc," TPC. (), URL: https://www.tpc.org/tpcc/ (visited on 05/04/2024).

[42] "Retwis," Gitbub. (), URL: https://github.com/antirez/retwis (visited on 05/04/2024).

[43] "Nics," DPDK. (), URL: https://core.dpdk.org/supported/nics/ (visited on 05/04/2024).

[44] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "AccelTCP: Accelerating network applications with stateful TCP offloading," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 77–92, ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/moon.