# Practical DB-OS Co-Design for Modern, High-Performance Database Engines

by

Xinjing Zhou

B.E., Hangzhou Dianzi University (2017)
M.S., Zhejiang University (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026



Authored by: Xinjing Zhou
Department of Electrical Engineering and Computer Science
May 15, 2026

Certified by: Michael Stonebraker
Adjunct Professor Emeritus of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

# THESIS COMMITTEE

## Thesis Supervisor

**Michael Stonebraker**
*Adjunct Professor Emeritus of Electrical Engineering and Computer Science*
*Department of Electrical Engineering and Computer Science, MIT*

## Thesis Readers

**Samuel Madden**
*MIT College of Computing Distinguished Professor*
*Department of Electrical Engineering and Computer Science, MIT*

**Viktor Leis**
*Professor*
*School of Computation, Information and Technology, Technical University of Munich*

# Practical DB-OS Co-Design for Modern, High-Performance Database Engines

by

Xinjing Zhou

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2026 in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

**ABSTRACT**

Database management systems (DBMSs) rely on operating systems (OSes) to access hardware resources, but the abstractions exposed by general-purpose OSes are often poorly matched to database workloads. DBMSs are optimized to process data-intensive workloads, while OSes provide interfaces that emphasize security, protection, and fair multiplexing of hardware resources. This long-standing mismatch has become more severe as modern hardware evolves rapidly. Networks and storage devices have improved by orders of magnitude, while single-core CPU performance has improved much more slowly. As a result, operating systems increasingly dominate query and transaction execution time. Existing approaches such as full kernel bypass, custom kernels, and kernel modules can reduce this overhead, but they often sacrifice key practicality properties, including compatibility, security, deployability, and maintainability.

This thesis argues for *practical DB–OS co-design*, replacing poorly matched OS mechanisms on the database critical path while preserving these practicality properties. It begins with a whole-stack measurement study of memory-optimized OLTP systems, which identifies the OS/kernel stack as a major source of cost under modern hardware and workloads. Guided by the principle of specializing only the mechanisms that limit performance, and doing so at the lowest sufficient privilege level, the thesis develops three systems. Tux is a database-aware networking stack that achieves kernel-bypass-class performance without requiring user-space NIC drivers or exclusive device ownership. Calico is a DBMS-controlled buffer pool that matches hardware-accelerated in-memory translation while preserving DBMS control over eviction and scalable I/O. Libdbos leverages hardware virtualization to safely expose privileged mechanisms to the database process, enabling instantaneous virtual-memory snapshots and efficient buffer management.

Together, these systems show that the DB–OS interface can be redesigned around database semantics across communication, memory management, and privilege boundaries. The result is a practical path to large end-to-end performance gains without giving up the practicality properties required by production database systems.

Thesis supervisor: Michael Stonebraker
Title: Adjunct Professor Emeritus of Electrical Engineering and Computer Science

# Acknowledgments

I would like to begin by thanking my advisor, Mike Stonebraker. Mike has been incredibly supportive of my research and has given me the freedom to explore the intersection of databases and operating systems on my own terms. When I started my PhD, I felt a bit nervous because of his stature as a Turing Award winner, but I quickly learned that he welcomes open discussion and thoughtful debate. Those conversations pushed me to think much more deeply, and they have been instrumental in shaping how I do research. Working with him also expanded my vocabulary of slang and idioms. One phrase he uses often is “finding the high pole in the tent” and over the years, I found many "poles" (key bottlenecks) in many "tents" (research problems) that the phrase quietly became a guiding principle for this thesis.

I am grateful to my committee members, Viktor Leis and Sam Madden. Viktor has been a real role model for me in how to do good research and how to write good papers. Conversations with him always feel effortless, and I often come away with a clearer perspective. I sometimes feel like my research has become a bit more German because of those discussions. Sam has also been incredibly supportive throughout my time at MIT. He is unfailingly kind, his advice has improved this thesis in many places, and he quietly does an enormous amount of work behind the scenes to keep our lab running.

I have also been fortunate to learn from a remarkable set of mentors. Xiangyao Yu has been involved in almost every project I have worked on during my PhD, and I have learned an incredible amount from him about how to approach research thoughtfully and rigorously, and especially about asking the “why now” questions. Goetz Graefe has been generous with his suggestions, feedback, and historical perspective, particularly on some of the earlier projects. And I want to thank Andy Pavlo for inspiring me to pursue database systems research in the first place. I am glad we had the chance to collaborate on Calico, perhaps in part because of his enthusiasm for `mmap`, and working with Andy has always been a lot of fun. I would like to thank my internship mentors at Microsoft Research: Badrish, Phil Bernstein, Enrique Saurez, and Brian Kroth. I have learned a lot from them.

I am also grateful to my collaborators. Xiangpeng Hao, Jinming Hu, and Wenjie Xi have been fantastic to work with. I also want to thank Qian Li and Peter Kraft for the opportunity to collaborate on the DBOS project.

The Data Systems Group at MIT is probably the best group of people I could have hoped to spend my PhD with. Beyond the research itself, what I will remember most are the moments we shared together, the ski trips and outings, the dinners, the random conversations in the office, and especially the daily lunches that produced an absurd number of good memes. Research can be stressful and isolating at times, and this group made the whole journey feel much more fun. Thank you to Amadou, Andreas, Aniruddha, Anna, Brit, Chunwei, Darryl, Dominik, Elkindi, Emanuel,

Eugenie, Fabian, Ferdi, Geoffrey, Gerardo, Ibrahim, Jason, Jialin, Joana, Kapil, Laurent, Lei, Markos, Matthew Russo, Matthew Perron, Oscar, Pascal, Peter Chen, Ryan, Siva, Sushrut, Sylvia, Tianyu, Wenjia, Yuze, Zhuohan, and Ziniu.

Finally, I want to thank my dear wife and best friend, Ursula. Thank you for being supportive and caring for me through every high and low of my PhD journey. This would not have been possible without you. To my cat Tiandou, thank you for being the best companion. And to my parents and my brothers, thank you for the constant love and encouragement from across the ocean that kept me going through all of this.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Database systems are the backbone of modern enterprises, underpinning everything from real-time transaction processing and analytics dashboards to the persistent memory behind recommendation and search systems. The demands on them are also growing exponentially in both scale and diversity. Worldwide data generation is projected to exceed 220 zettabytes by 2026 [1]. Compounding this pressure, the rise of AI has introduced an entirely new class of workloads. These include retrieval-augmented generation (RAG) pipelines [2], AI agents that use databases as persistent memory, and semantic search over billions of high-dimensional embeddings via approximate nearest neighbor (ANN) indexes [3–5]. These workloads have access patterns that differ fundamentally from traditional workloads. To keep pace with the rapid growth of both use cases and data volumes, modern database systems must continue to push the boundaries of performance.

On the other hand, the hardware landscape underneath these systems has changed drastically over the last decade. I/O hardware performance has grown by an order of magnitude in the last decade, with network bandwidth rising from 10 Gbps to 600 Gbps [6,7] and PCIe Gen5 SSD arrays delivering tens of millions of I/O operations [8], while per-core CPU performance has barely moved and Moore's-law cost-performance gains have slowed down significantly [7]. The OS layers between the database and the hardware, once a negligible fraction of execution time, now dominate it. Saturating a 400 Gbps NIC or an array of 8 NVMe SSDs through the Linux kernel stack would consume all the CPU cores on a modern machine and leave essentially no cycles for query or transaction processing [8]. Our own measurements (Chapter 2) confirm the same shift inside production database engines. Communication overhead consumes 52–68% of CPU cycles in modern memory-optimized OLTP systems [9], and OS-level memory management consumes up to 75% of CPU time in larger-than-memory vector search. The DB–OS interface, not the engine internals, is now the first-order bottleneck.

## 1.1 Existing Approaches and Why They Fall Short

The mismatch between database needs and OS interfaces is not new [10]. Prior work has explored many forms of DB-OS co-design, ranging from custom database-specialized operating systems [11,12], to user-space kernel bypass for networking [13–17] and storage [18], to unikernels that compile the database with a library OS [19,20], to in-kernel specialization through kernel modules [21–23]. Despite this breadth of effort, none of these approaches have been adopted at

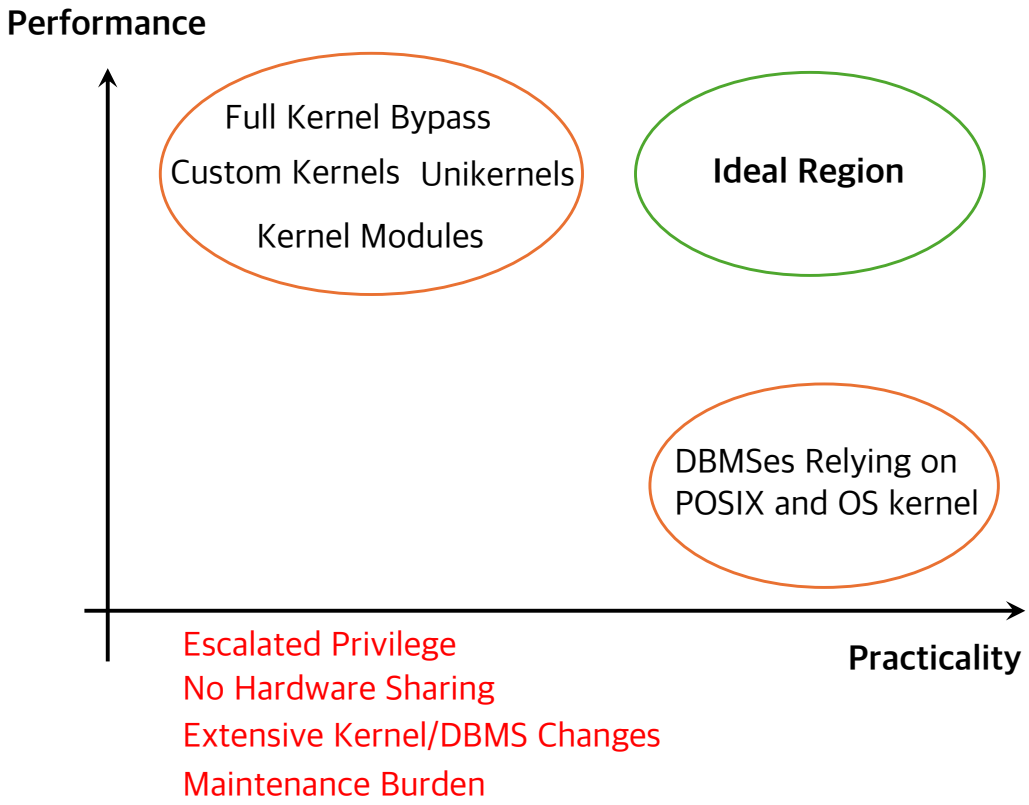



Figure 1.1: **The DB–OS co-design space** - Living with the kernel preserves practicality but leaves performance on the table; user-space kernel bypass recovers performance but sacrifices device sharing and deployability; custom kernel or kernel modules trade security and maintenance costs for performance.

scale by production database systems, which still face an unsatisfying choice along three axes: live with the kernel and pay the overheads quantified above, bypass the OS in user space and lose the deployability that made general-purpose OS adoption attractive in the first place, or modify the kernel itself and inherit a different set of security and maintenance costs. Figure 1.1 maps these approaches in the performance–practicality plane. We examine each in turn.

**Relying on general-purpose OS abstractions leaves performance on the table.** The OS provides general-purpose abstractions that are a poor fit for database workloads. Databases nonetheless accept these abstractions for good reasons: *portability* across operating systems and CPU architectures; *compatibility* with the surrounding ecosystem of containers, schedulers, observability agents, backup tools, and security scanners that all assume standard OS semantics; *isolation* from co-located processes, since the kernel mediates address spaces, file descriptors, and device access so that a bug or compromise in one process cannot reach another; and *operational simplicity*, since standard deployment, monitoring, and debugging tools work out of the box. The cost of this practicality is the overheads quantified above. `mmap` cedes eviction and I/O control entirely to the kernel [24], hiding access patterns and triggering unscalable synchronization on multi-core systems; user-space hash-table buffer pools [25,26] preserve DBMS control but

pay significant software translation overhead on every page access; `fork`-based snapshots incur high latency from page-table copying and copy-on-write faults [27]. Engine-side optimizations from H-Store [28], VoltDB [29], HyPer [30], LeanStore [31], Silo [32], and SAP HANA [33] have squeezed buffer management, concurrency control, latching, and recovery, but cannot remove overheads that live below the system-call boundary.

**Bypassing the OS in user space sacrifices practicality.** One line of work bypasses the OS for specific resources by moving I/O entirely into user space. For networking, DPDK [13] and user-space TCP stacks [14–17] move the entire network stack into user space, communicating directly with NIC hardware. For storage, SPDK [18] provides direct access to NVMe devices. Unikernels [19] and custom OS designs take this further, compiling the database system with a library OS to eliminate the kernel altogether [20]. These approaches recover the lost performance but sacrifice practicality in multiple ways. First, they require exclusive device ownership, since DPDK takes control of the NIC, making it invisible to the host OS and incompatible with standard networking tools, multi-tenant deployments, and container orchestration. Second, they demand significant rewrites, since applications must implement their own TCP state machines and adopt polling-based execution models. Third, they suffer from ecosystem fragility, since DPDK's API/ABI breaks frequently across releases [34], forcing developers to choose between new features and code stability. ScyllaDB promotes DPDK compatibility in its Seastar framework but does not deploy it in production [35]. To our knowledge, Yellowbrick [36] is the only database vendor that makes significant use of kernel-bypass, and it required writing custom network and storage drivers.

**Modifying the kernel inherits a different set of costs.** A complementary line of work specializes the kernel itself for databases. Kernel modules extend or replace specific subsystems, for example the virtual-memory machinery behind the buffer pool or the I/O path that feeds analytical scans [21–23]. These designs sidestep the device-ownership problem of DPDK/SPDK, but pay other costs. Kernel modules run with full kernel privilege, so any bug in the module becomes a system-wide security and reliability hazard rather than a contained application crash; they also live outside the upstream kernel and must be maintained against an internal interface that the kernel community is free to change at any time, with no stability guarantees. The database is once again coupled to a specific kernel interface that evolves on its own schedule, eroding the deployability that made general-purpose OS adoption attractive in the first place.

In short, the database community faces a dilemma. Either accept OS overhead on the critical path, or abandon or modify the OS and pay the engineering, security, and compatibility costs. Neither option is satisfactory for production database systems that need both high performance and broad deployability.

## 1.2 Practical DB-OS Co-Design

This thesis proposes a different path, *practical DB-OS co-design*, which closes the gap between the database and the hardware without forking or patching the host kernel. The central insight is that a new generation of OS interfaces and hardware support has made this newly tractable. Together, they expose capabilities that previously required a kernel patch or full device takeover. Combined

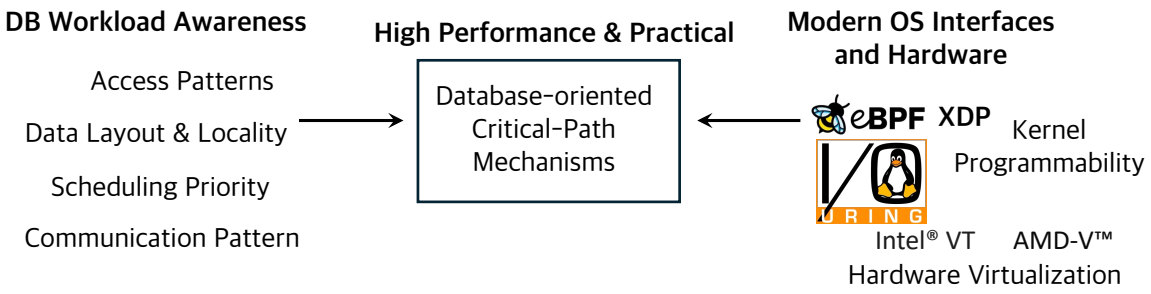



Figure 1.2: **Practical DB-OS co-design** - Databases can combine their workload knowledge with modern OS interfaces and hardware to make co-design practical and performant.

with the rich semantic knowledge that databases have always possessed but could not safely act on, they open a new region of the design space that was unavailable to prior work.

**New OS interfaces and hardware enable safe specialization without kernel modifications.** eBPF and XDP let verified application logic run inside the kernel without a custom module, making it possible to intercept and reshape I/O paths with database-specific logic. `io_uring` exposes asynchronous, batched I/O that amortizes system-call overhead and supports device passthrough. Hardware virtualization (Intel VT-x, AMD-V) can safely raise a single user process to kernel-space privilege. Each of these mechanisms removes a category of bottleneck that previously had no safe path to specialization.

**Database semantics make specialization effective.** Databases already know facts the OS cannot see: which pages belong to which objects, which accesses are likely to recur, which requests are latency-critical, and which work can be batched or reordered. Practical co-design uses this knowledge to decide where the OS boundary should be specialized and what each specialized mechanism should do. The interfaces and hardware support above make these mechanisms deployable.

Putting the two together yields the design pattern of this thesis: *database-oriented mechanisms on the critical path*, built by exploiting database semantics and modern kernel interfaces and hardware, and deployed at the lowest privilege level that removes the bottleneck. Figure 1.2 illustrates the pattern. Two operational principles follow.

1. **Critical-path specialization.** For each performance-critical interaction with the OS, the database introduces a purpose-built mechanism whose semantics are matched to database workload, leaving the rest of the OS in place.

2. **Minimum privilege.** Each new mechanism is implemented at the lowest privilege level that works: user space when possible, safe kernel extensions (eBPF/XDP) when user space is insufficient, and hardware-level privilege via virtualization only when nothing else suffices. The host kernel source is never patched and no production kernel subsystem is taken offline, preserving the ecosystem of drivers, tools, and security mechanisms that production systems rely on.

The core statement of the thesis is as follows:

> **Thesis Statement.** *Modern databases should replace only the OS mechanisms on their critical path with database-oriented designs, using the lowest privilege level sufficient to remove each bottleneck. This form of practical DB-OS co-design delivers large end-to-end performance gains while preserving the compatibility, security, and deployability of general-purpose operating systems.*

## 1.3 Contributions

This thesis makes the following contributions:

**Revisiting OLTP Through the Looking Glass after 16 Years (Chapter 2).** We start by revisiting the classic OLTP bottleneck analysis with a systematic whole-stack performance breakdown of VoltDB on modern hardware. We show that communication overhead now dominates and establish that the OS boundary, not the engine internals, is the critical-path bottleneck in modern OLTP engines. This study supplies the empirical foundation for practical DB–OS co-design.

**Partial Kernel-Bypass Networking for Databases with Tux (Chapter 3).** Following the measurement study, we attack the communication bottleneck. Tux is a database-aware high-performance networking stack without giving up compatibility. Database communication is often message-oriented, yet existing stacks force it through a byte-stream TCP interface that adds framing, copying, and head-of-line blocking on the hot path. Additionally, the kernel NIC driver is not the bottleneck; the generic socket and TCP layer above it is. Tux closes this gap by leveraging modern OS interfaces, eBPF and XDP, to bypass the kernel TCP stack while keeping the vendor-supported NIC drivers in place, preserving compatibility with the surrounding OS networking ecosystem. Tux introduces a message-based transport that decouples reliability from in-order delivery and preserves message boundaries without copy overhead, a pushdown abstraction that runs DBMS logic directly on the network cores and thereby enables database-aware scheduling that existing kernel-bypass stacks lack, and a library (Libtux) offering zero-change and minimal-change deployment modes. Evaluated on Redis, Memcached, VoltDB, and LeanStore, Tux improves throughput by up to 2.3× and reduces p99 latency by up to 4.7× over state-of-the-art kernel-bypass systems, with no exclusive device ownership. Tux shows that the communication bottleneck can be practically removed by combining database semantics with a kernel programmability interface.

**User-Space High-Performance Buffer Management (Calico, Chapter 4).** With communication addressed, the next OS interface on the critical path is memory management, which dominates CPU time in larger-than-memory workloads such as vector search. Calico is a DBMS-controlled buffer pool that matches `mmap`-class in-memory translation speed while keeping full control over eviction and I/O in user space. The key observation is that database pages are organized hierarchically and index accesses exhibit strong spatial locality. This regularity inspires Calico's multi-level array-based translation layer, together with path caching, active hole punching for reclaiming cold translation memory, and group prefetch for asynchronous I/O. Integrated into PostgreSQL/pgvector, Calico delivers up to 3.95× speedup for in-memory HNSW vector

search and 6.57× for larger-than-memory workloads, with no kernel modifications and no elevated privilege.

**Privileged Kernel Bypass via Hardware Virtualization with LibDBOS (Chapter 5).** Tux and Calico together cover the bottlenecks that user space can reach. A residual class of bottlenecks remains, namely virtual-memory mechanisms that require direct control of page tables, TLBs, and interrupts in kernel space. Specializing these mechanisms with DBMS semantics yields far better primitives for database use cases than the general-purpose ones the kernel exposes through POSIX, but a database has no safe way to reach the privileged hardware. LibDBOS addresses this last category by escalating privilege only when nothing weaker suffices. Hardware virtualization, exposed to a single database process through a hypervisor, makes that privilege available without patching the host kernel or replacing production kernel subsystems. Building on this foundation, LibDBOS enables Snappy, which provides instantaneous virtual-memory snapshots (21× faster than `fork`, reducing Redis checkpoint tail latency by 20×), and Tabby, a buffer pool that leverages virtual-memory hardware for `mmap`-speed translation without TLB shootdowns while preserving DBMS eviction control.

The three systems together explore different points on the privilege spectrum of practical co-design, from pure user space (Calico), to partial kernel bypass (Tux), to hardware-level privilege (LibDBOS), and show that the DB–OS interface can be redesigned around database semantics across communication, memory management, and the privilege boundary itself, without giving up the compatibility, security, or deployability that production database systems demand. Chapter 6 returns to this design space and discusses numerous future directions opened by this thesis.

# Chapter 2

# OLTP Through the Looking Glass 16 Years Later

## 2.1 Introduction

Chapter 1 argued that the DB–OS boundary, not only database engine internals, has become a first-order bottleneck in modern database systems. This chapter makes that argument concrete for memory-optimized OLTP engines by revisiting the 2008 "Looking Glass" study [37] under today's hardware, workloads, and isolation models.

The original study explored OLTP performance on conventional hardware and software [37]. Notably, the study ran TPC-C on the Shore DBMS, a conventional DBMS with disk-based storage, a buffer pool, concurrency control based on locking, and standard Aries-style write-ahead logging. As such, it was typical of the commercial DBMSs of the time. The study found that the overwhelming majority of CPU time was spent on services such as buffer pool management, concurrency control, crash recovery, and latching. Hence, to go a lot faster, one has to deal with these sources of overhead.

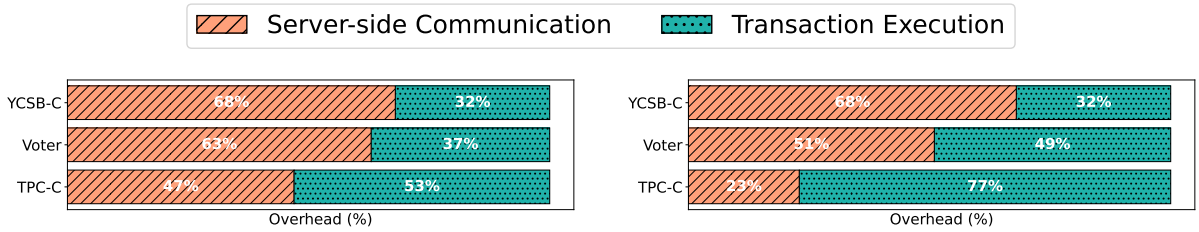
Since then, there have been several high-performance OLTP engines, including H-Store, VoltDB, Hyper, LeanStore, MemSQL, Silo, and HANA. These systems exploit various innovative solutions to the issues noted above to provide tremendous performance improvements over traditional systems in benchmarks. However, most academic OLTP benchmarks (including [37]) have two major problems.

**Ignored Network/Communication**. Previous work [37] only analyzed back-end components of a database system and ignored the messaging cost of communicating with a front-end application or within the database system. Even when running a stored procedure (SP) transaction model, this cost cannot be ignored. Moreover, although stored procedures are standard practice in research systems, many real-world applications prefer running transaction logic on the client side for better debuggability, security, and ease of version control [38]. This makes messaging costs much more expensive than in the SP model.

**Ignored User-Defined Code Isolation**. When systems use an SP transaction model, most assume it is permissible to run stored procedures in the same address space as the DBMS. This means there is no *isolation* between an SP and the DBMS kernel or between different SPs. This allows errant or malicious SPs to interfere with each other and the DBMS. Worse yet, malicious

SPs can obtain unauthorized data, which presents a security flaw. In some use cases, this risk is acceptable due to careful testing; in others, non-isolation is unacceptable. As cloud database systems move toward a multi-tenant architecture, such risks will become less acceptable.

In response, we present “Looking Glass 2.0”. We perform whole-stack benchmarks measuring throughput and response time on modern hardware, comparing the SP model with client-side transaction logic under different levels of isolation. To avoid the performance bottlenecks of traditional DBMSs, we primarily focus on high-performance OLTP engine VoltDB in this study. PostgreSQL results are also included for some minor experiments for reference.




(a) **Client-Side Transaction Model** (b) **Stored-Procedure Transaction Model**

Figure 2.1: **Server-side CPU Overhead Breakdown** - In the client-side transaction model the DBMS only processes SQL queries, while in the stored-procedure model it additionally runs the transaction logic.

Figure 2.1 depicts one of the major findings of this study. When isolation is not required, the CPU cycles spent in messaging are "the high pole in the tent", even when using a stored procedure model. With a traditional transaction model, CPU overhead in communication is higher. While the stored-procedure model helps reduce such costs by reducing the number of round trips between transaction logic and the database engine, it does not help with simpler transactions such as those in YCSB-C and Voter workloads. Hence, any improvements to DBMS performance impact a minority of the path length, and the most profitable source of OLTP improvement is therefore communication. When isolation is required, communication cost is dramatically the highest cost item, overshadowing the CPU cycles spent in transaction execution.

Hence, we explore possible improvements to TCP/IP and Linux kernel. These include kernel bypass with user-space networking for messages between the client and the DBMS. We also report unpleasant experiences when applying kernel-bypass techniques to DBMSs, suggesting room for improvement on the OS side. To isolate SPs from the DBMS, we classified five possible levels of isolation and explored a spectrum of approaches from no isolation to virtual machine isolation. Higher degrees of isolation resulted in additional messages and higher costs.

| Database System | Programming Language for Stored Procedures and UDFs | Isolation Mechanisms | | | |
|---|---|---|---|---|---|
| | | None | Language Isolation | OS-Level Isolation | Virtualization |
| VoltDB | Java | | ✓ | | |
| Oracle | PL/SQL, Java, JavaScript | | ✓ | | |
| SQL Server | T-SQL, C# | | ✓ | | |
| PostgreSQL | PL/pgSQL, C, Python, Rust, JavaScript | ✓ | ✓ | | |
| MySQL | PL/SQL, JavaScript | ✓ | ✓ | | |
| IBM DB2 | PL/SQL, Java, C++ | ✓ | ✓ | | |
| MariaDB | PL/SQL, C/C++ (UDFs) | ✓ | ✓ | | |
| SAP HANA | PL/SQL, R | | ✓ | | |
| Teradata | PL/SQL | | ✓ | | |
| MongoDB | JavaScript | | ✓ | | |
| Snowflake | Java, JavaScript, Scala, Python | | ✓ | ✓ | |
| Redshift | PL/pgSQL, Python, AWS Lambda | | ✓ | ✓ | ✓ |

Table 2.1: **Programming languages and isolation mechanisms for user-defined code in existing database systems**

The rest of this chapter develops this argument in three steps. First, it presents a whole-stack performance breakdown of a modern OLTP engine and identifies where CPU time goes under different transaction execution models. Second, it examines the design space for isolating user-defined code in OLTP systems and quantifies the cost of stronger isolation. Finally, it uses these measurements to motivate the DB–OS co-design directions explored in the later chapters.

## 2.2 Background and Methodology

We first examine isolation mechanisms before explaining our benchmarking methodology.

### 2.2.1 Isolation Mechanisms for User Code

**Client-Server Isolation**. The baseline option is to run user-defined code in a client process separate from the DBMS. This is also commonly denoted as *interactive transactions* and is supported by virtually all DBMSs. The extreme case of isolation is to place the user and server processes on different servers connected through the network. Reducing communication overhead requires co-locating the user code and DBMS on the same server, as we discuss next.

**No Isolation**. Many systems, including PostgreSQL, DB2, MariaDB, and Teradata, allow user-defined logic in *unsafe* languages such as C by loading user code into the DBMS as a shared library. Given that malicious or erroneous user-defined code can easily access the content of DBMS memory

or bring down the entire system, this approach provides effectively *no isolation*. In exchange for giving up isolation, this maximizes performance as user-defined code can communicate with the DBMS kernel through shared memory.

**Language Isolation.** The next level of isolation is provided by *safe* stored procedure (SP) languages running within the DBMS process. Such languages include domain-specific languages (e.g., PL/SQL, T-SQL), virtual-machine-based languages (e.g., Python, Java, JavaScript), and memory-safe languages (e.g., Rust). By disallowing unsafe memory accesses, safe SP languages provide some level of isolation. However, the isolation fully relies on the correctness of compiler and language runtime and is therefore somewhat limited. For example, Oracle [39] and VoltDB run Java-based stored procedures in the same address space as the DBMS kernel. User-defined code can break out of the virtual machine by using unsafe language features, VM exploits [40], or bugs in the compiler.

**OS-Level Isolation**. To provide better isolation guarantees than language-based isolation, one can run an SP in a different OS process than the DBMS but on the same server. However, important operating system resources such as the file system and network devices are still shared. One way to provide stronger isolation is using *containers*, which virtualize all OS resources, and Linux's *SECCOMP* mechanism, which limits the system calls that can be made by a process. Snowflake, for example, runs user-defined code in a separate container with access to a dedicated file system without the ability to make network connections. In this approach, user-defined code uses OS-provided inter-process communication (IPC) mechanisms such as sockets to interact with the DBMS kernel process. This provides much better isolation than language-based approaches.

**Virtualization**. OS-level isolation obviously relies on the correctness of the OS. Attacks against the OS kernel can compromise the memory of all running processes. Therefore, a stronger isolation approach is to run user-defined code on a separate OS kernel in a virtual machine on the same host. This confines malicious code execution to the virtual machine. This model requires inter-VM communication between the DBMS and user-defined code.

To summarize, the five mechanisms discussed provide trade-offs between the proximity of the user code to the DBMS and the degree of isolation and communication overhead. Table 2.1 summarizes the isolation mechanisms used for user-defined logic in existing DBMSs. Note that the table is non-exhaustive.

### 2.2.2 VoltDB Primer

To understand our testing approach, we first review VoltDB's architecture. As shown in Figure 2.2, VoltDB adopts a shared-nothing architecture, partitioning data across multiple cores. Each core has its own partition serviced by a single worker thread. Networking threads handle client connections and interact with the TCP/IP stack in the Linux Kernel. Each worker runs stored procedures (SPs) from a task queue in a single-threaded manner within the same process as the DBMS kernel. The Java-based query engine executes SPs using an in-memory C++ storage engine, and SPs are written in Java with embedded SQL. VoltDB distinguishes between single-partition and multi-partition transactions. Single-partition transactions, which access data within one partition, are optimized to run to completion without locks or latches. As a result, VoltDB maximizes CPU efficiency in its OLTP engine when executing these transactions. Conversely, multi-partition transactions are serialized through a global coordinator, allowing only one at a time, and use standard two-phase commit. In this work, we focus on single-partition transactions, as they provide state-of-the-art

performance, which we strive to improve on. VoltDB separates the logic of transaction processing from network I/O activities. A group of dedicated network threads are created for interfacing with the Linux Kernel with BSD socket APIs and event polling. The network threads parse requests and dispatch stored procedure invocations to dedicated OLTP workers through message passing. The network threads also write responses back to the sockets after OLTP workers finish transaction execution.

### 2.2.3 Network and Isolation Modeling

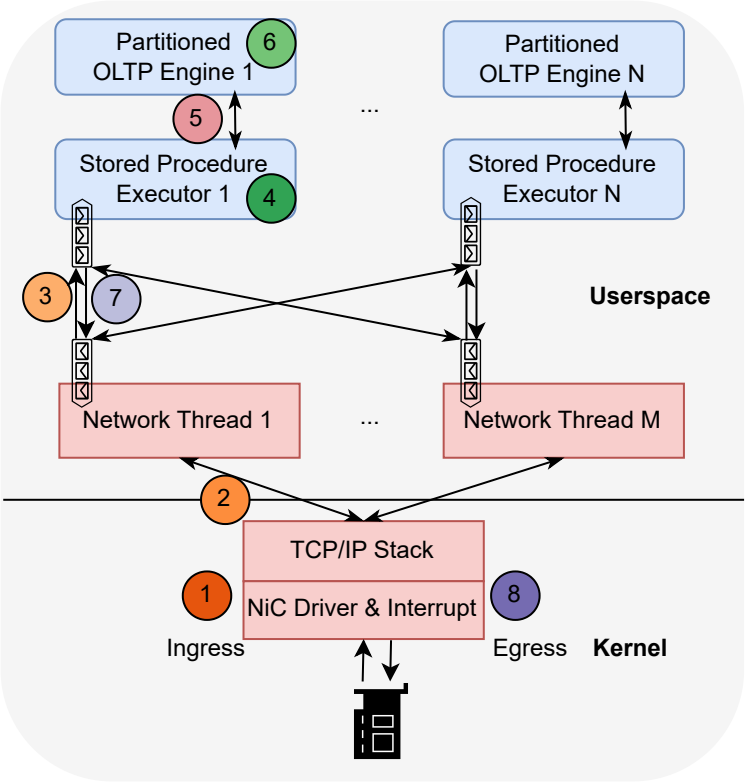



Figure 2.2: **Architecture of VoltDB**

**Interactive Transactions**. Since VoltDB requires a stored procedure model, we simulate interactive transactions by breaking any stored procedure into multiple stored procedures, one per SQL statement. This closely simulates a classic environment with the client running on the same machine as the DBMS. As an optimization, we batch together SQL queries that have no dependencies. These batches are then run as individual stored procedures.

**User-Space Networking**. Like most DBMSs, VoltDB relies on the TCP/IP stack of the host operating system. We extended VoltDB with support for user-space networking using DPDK and FStack [16], a user-space TCP/IP stack. In order to interface with F-stack, we had to rewrite major parts of the VoltDB networking layer because F-stack requires using a set of new polling/read/write APIs and only supports a single-threaded TCP/IP stack. Therefore, F-stack-enabled VoltDB uses only one network thread for performing network I/O. The network thread spin-polls the NIC for available packets.

**Language Isolation**. Java SPs for VoltDB are compiled into individual classes and loaded into the same JVM on which VoltDB runs. Therefore, VoltDB provides language isolation by default.

**OS-Level (Process) Isolation**. To provide process isolation, we modified the VoltDB SP runtime to pair every VoltDB worker with a separate Linux process in which SPs for that worker are loaded. To communicate between this process and the corresponding worker, we compared three IPC mechanisms: TCP/IP, shared memory with busy-polling for notification, and shared memory with a Unix domain socket for notification. When exchanging messages between the SP process and the OLTP engine, SQL queries and result sets are serialized and deserialized. Although we could have run each SP in its own process, we chose not to, as the performance would be essentially identical to the architecture we tested.

**Containerization**. In addition to process isolation, we explored running each SP process in a Docker container and the same set of communication mechanisms used for process isolation.

**Virtualization**. The last and strongest isolation approach we explored is virtualization. We ran each SP process on top of a Linux kernel in a virtual machine (KVM). For this setup, the DBMS kernel communicates with each SP process through the TCP/IP stack in the host Linux kernel as well as the guest Linux kernel. When a network packet is sent from the DBMS kernel, it needs to first traverse the TCP/IP stack of the host Linux kernel, and then it is routed through the kernel network bridge to arrive at a virtualized NIC maintained in the hypervisor in the kernel. Lastly, the packet crosses the virtualization boundary to be handled by the receiving side of the guest kernel's virtual NIC and TCP/IP stack.

### 2.2.4 Profiling Methodology

We use manual instrumentation to profile VoltDB and the Linux perf tool for the kernel. Specifically, for kernel-space profiling, we leveraged the `perf-trace` [41] tool to place trace points inside the Linux kernel, which replace traced instructions with breakpoint instructions. When a trace point is hit, the Linux kernel traps into a breakpoint handler that records timing information and then resumes execution. For VoltDB profiling, we found `perf-trace` [42] too expensive as each activated trace point results in a context switch to the kernel. Instead, we manually instrumented the VoltDB code to record a trace entry in a memory buffer. At the end of the benchmark, VoltDB dumps the buffer to a file in the same format as the output by `perf-trace`.

## 2.3 Experimental Evaluation

### 2.3.1 Environment

**Hardware**. We ran all experiments on Google Cloud using two compute instances, each with 16 vCPU (Intel Haswell, 2.3GHz) and 64 GB memory. The two instances are connected by a network with a measured 16-Gigabit bandwidth in one direction and 60us median round-trip latency (measured using sockperf).

**Software**. The machines are running Linux version 6.8.0-1007-gcp. VoltDB is compiled with OpenJDK version 1.8.0_402 and gcc 11.4.0. We configure the client machine to run 512 concurrent connections to the VoltDB server. Each client iteratively runs the benchmark script. We evaluate

system performance using three workloads. As shown in Table 2.2, these workloads cover a wide range of transaction complexity and round-trips to the database system.

| Workload - TX | # DB Interactions | Ratio |
|---|---|---|
| YCSB C - Get | 1 | 100% |
| Voter - Vote | 2 | 100% |
| TPC-C - New Order | 8-18 | 45% |
| TPC-C - Payment | 5 | 43% |
| TPC-C - Delivery | 4 | 4% |
| TPC-C - Stock Level | 2 | 4% |
| TPC-C - Order Status | 3 | 4% |

Table 2.2: **Workload Descriptions** - A DB interaction is the largest batch of SQL queries that can be sent to the DBMS in one round-trip to be executed independently. We break the logic of each transaction into the shortest sequence of interactions with the database systems where an interaction depends on the result sets of previous interactions.

**YCSB**. The Yahoo! Cloud Serving Benchmark is a key-value store benchmark. We initialize the YCSB table with 10M 128-byte key-value pairs and perform random single-tuple index lookups.

**Voter**. This benchmark simulates a phone-based election process and contains a collection of single record queries and updates. 10,000 voters are given a fixed number of votes for 6 candidates. Each voter selects any candidate iteratively until their votes are exhausted.

**TPC-C**. The TPC-C benchmark is the industry-standard benchmark suite for OLTP databases. Each transaction contains complex interactions with the database. We configured the system with 256 warehouses and partitioned the database by warehouse. We modified the benchmark so that every transaction targets only a single warehouse to avoid multi-partition transactions on VoltDB.

### 2.3.2 Where do the CPU Cycles Go?

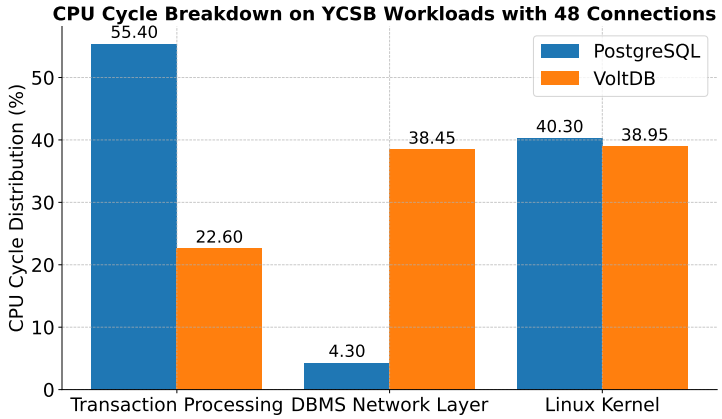


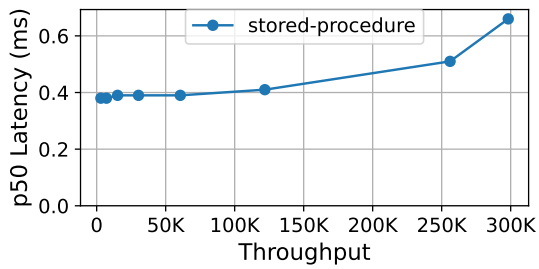

Figure 2.3: **CPU Cycle Distribution for PostgreSQL and VoltDB** - Cycles are classified into Transaction Processing, DBMS Networking Layer, and Linux Kernel.

We begin by showing CPU cycle distribution on the server. We run simple YCSB-C workloads with 48 connections. The results are shown in Figure 2.3. VoltDB spends less than a quarter of the total cycles on transaction processing. This is partly because the VoltDB eliminated buffer pool, locking, and latching that were bottlenecks pointed out by the original looking glass paper [37]. Instead, the CPU bottlenecks are now shifted to the DBMS networking layer and the Linux Kernel. Surprisingly, the DBMS networking layer in VoltDB contributes almost 40% of the CPU cycles. Our profiling found that 8% of the cycles are spent on `epoll_wait/epoll_ctl/read/write` functions in the glibc library. The rest of the cycles are mostly spent on message passing and scheduling among the network threads and OLTP workers. For reference, we also included PostgreSQL (v13) results. We configure PostgreSQL buffer pool so that all data fits in memory. This eliminates storage I/O overhead and allows us to focus on communication within the system. As expected, transaction processing dominates with 55% of cycles due to the bottlenecks of a traditional DBMS [37]. The DBMS network layer consumes only 4.3% in PostgreSQL compared to 39% in VoltDB. This is because transaction-processing worker directly read/write to/from a TCP socket, avoiding most of the DBMS internal communication overhead. Interestingly, there are still nearly 40% cycles spent in Linux Kernel on networking for PostgreSQL, similar to VoltDB.

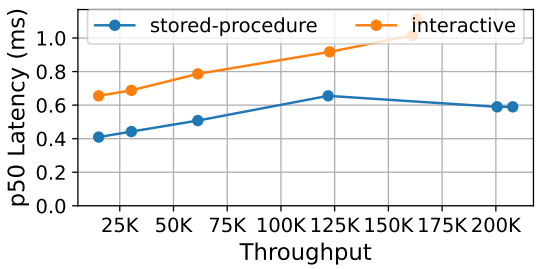
| Phase | Description |
|---|---|
| 1 Network Receive | Time from receiving data on the NIC to readying it in a socket. |
| 2 Socket Read | Time to read data from the socket. |
| 3 Request Queuing | Time to assign the request to a OLTP worker. |
| 4 Procedure Execution | Time to run the transaction logic. |
| 5 Isolation Overhead | Time for communication related to isolation. |
| 6 Query Execution | Time to run SQL queries. |
| 7 Response Queuing | Time to queue the response for transmission. |
| 8 Network Send | Time to write the response through the network stack. |

Table 2.3: **Phases of Transaction Processing in VoltDB**

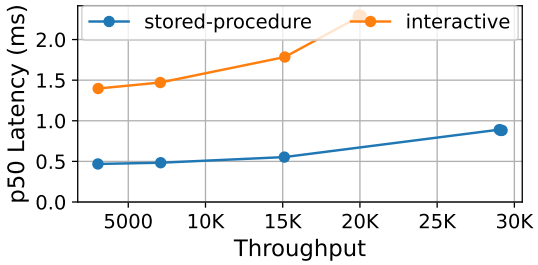
### 2.3.3 Results for No Isolation




(a) **YCSB**




(b) **Voter**

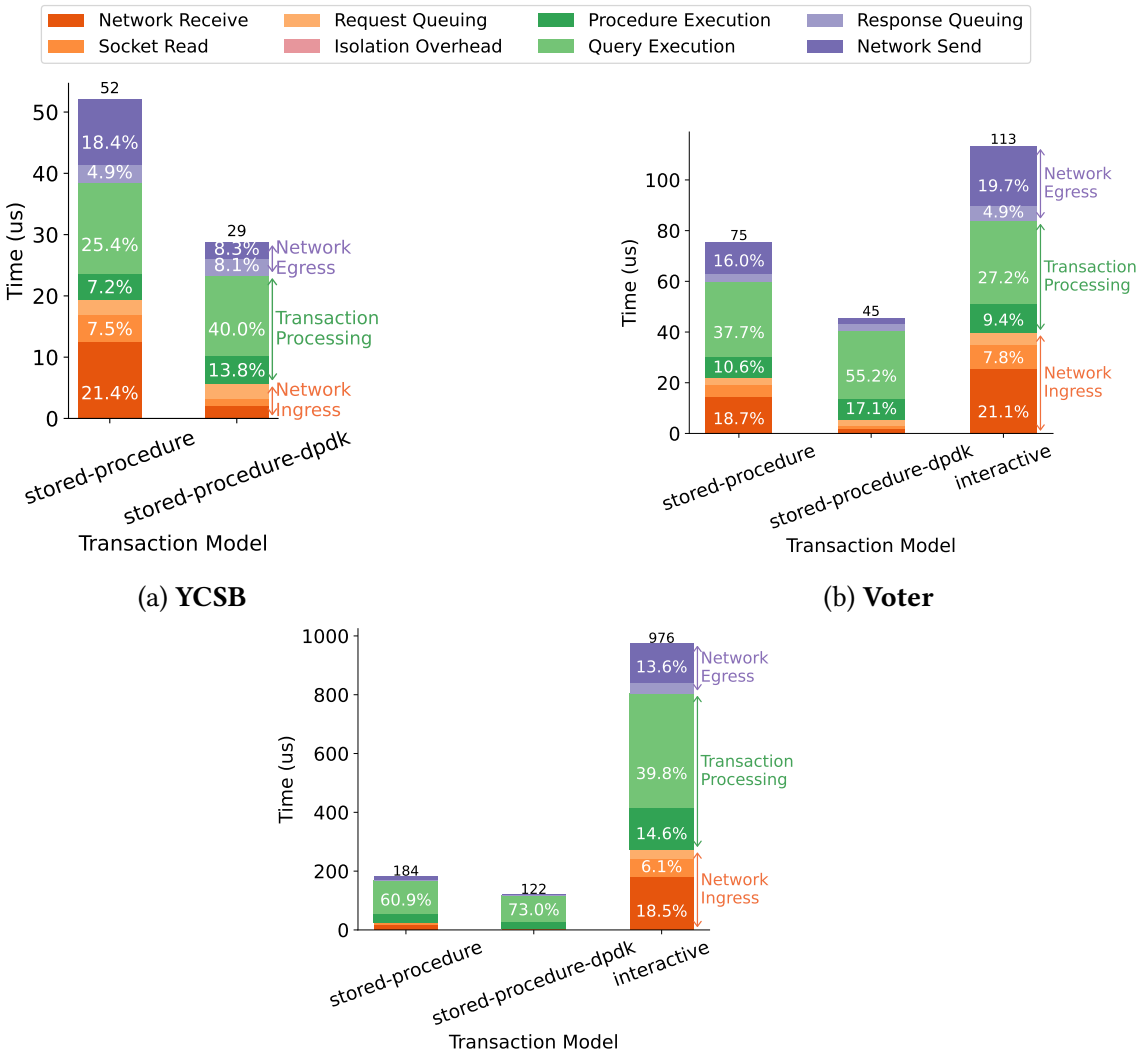



(c) **TPC-C**

Figure 2.4: **Median Latency vs. Load for Interactive Transactions and Stored-procedure Transactions**

We next present detailed results for the no-isolation case, comparing interactive transactions with SPs. Specifically, in Figure 2.4 we show the median latency for the three benchmarks as we ramp up the throughput on our system. Note that there is a single curve for YCSB as the two implementations are effectively identical. Also, note that the logic for both cases is the same; hence, the only difference between the two implementations is the number of messages. Obviously, throughput increases as the load is applied until our system becomes CPU-bound and saturates. Equally obvious is the increase in latency as load is applied because of queuing delays.

With the Voter workload, shown in Figure 2.4b, SP transactions have up to 72% lower latency and 23% higher achievable throughput. With more complex workloads such as TPC-C where there are more network round-trips, shown in Figure 2.4c, we observe that interactive transactions incur 3.5× higher latency at similar throughput points. The SP model can achieve a maximum throughput that is 2.1× higher than the interactive case. These results show the desirability of using an SP model.




(a) **YCSB**

(b) **Voter**

(c) **TPC-C**

Figure 2.5: **Server-side Transaction Latency Breakdown (Excluding Queuing)**

**Networking Dominates CPU Time.** Figure 2.5 presents a detailed breakdown of server-side latency for interactive (bars labeled as interactive) and SP transactions (bars labeled as stored-procedure) for our three benchmarks. Each bar represents the average CPU time spent in each of the eight categories measured in microseconds (µs). The first conclusion to draw from Figure 2.5 is the high pole in the tent is usually the server CPU time spent in networking. Except for TPC-C, networking dominates DBMS processing. Remarkably, it is more expensive to send work to the DBMS and receive answers than it is to construct those answers. Of course, if the transaction stored procedures get beefy enough, as exemplified by TPC-C, the system remains DBMS-bound. The second point is that stored procedures are a good idea. They lower the number of messages and increase the percentage of work spent on DBMS processing. The third point to make is that there is no magic bullet for improving network CPU time; the cost is spread over the whole stack. As a result, one needs to explore kernel bypass to get better performance, a topic to which we now turn.

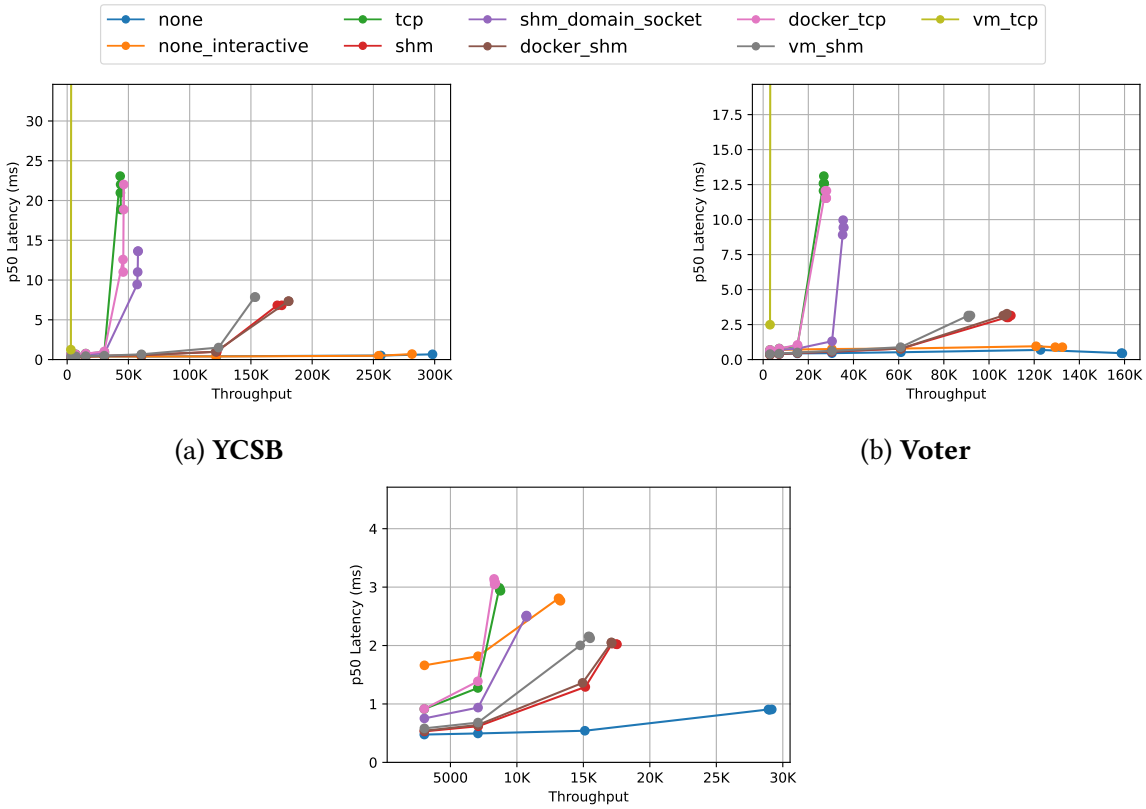
**Kernel Bypass.** To leverage kernel bypass, we replace the Linux networking stack used by VoltDB with a user-space TCP/IP stack (F-stack) using the DPDK library. As shown in Figure 2.5 (bars labeled `stored-procedure-dpdk`), with kernel-bypass and the SP model, the CPU time spent on transaction processing increases from 33%/48%/70% to 53.8%/72%/91% for YCSB/Voter/TPC-C workloads. These improvements mainly come from the elimination of system calls, reduced copying, interrupt processing, and a simpler networking stack. Note that the overhead of internal DBMS communication for queuing requests and responses remains.

**End-to-end Latency Breakdown.** We next break down transaction end-to-end latency. To minimize queuing delays distorting the results, we offer the maximum load before the end-to-end latency spikes. The results are shown in the following table, with the first three rows using the SP model and the last three rows using the client-side transaction model:

| Workload | Total | Cli.↔Srv. | Srv. Net. | Xact Exec. |
|---|---|---|---|---|
| YCSB-C | 410$\mu$s | 312$\mu$s | 64$\mu$s | 34$\mu$s |
| Voter | 425$\mu$s | 343$\mu$s | 40$\mu$s | 42$\mu$s |
| TPC-C | 483$\mu$s | 312$\mu$s | 42$\mu$s | 129$\mu$s |
| YCSB-C | 410$\mu$s | 312$\mu$s | 64$\mu$s | 34$\mu$s |
| Voter | 688$\mu$s | 572$\mu$s | 67$\mu$s | 49$\mu$s |
| TPC-C | 1,471$\mu$s | 710$\mu$s | 310$\mu$s | 451$\mu$s |

The results show that the time spent on client-to-server networking (the 3rd column) is the dominant component. This component increases further when using the client-side transaction model, as there are more round-trips. It also shows the superiority of the SP model. When running TPC-C using client-side model, the time for transaction execution increases compared to SP model due to repeated SQL parsing, planning, serialization, and deserialization.

### 2.3.4 Results for Isolation




(a) **YCSB**

(b) **Voter**

(c) **TPC-C**

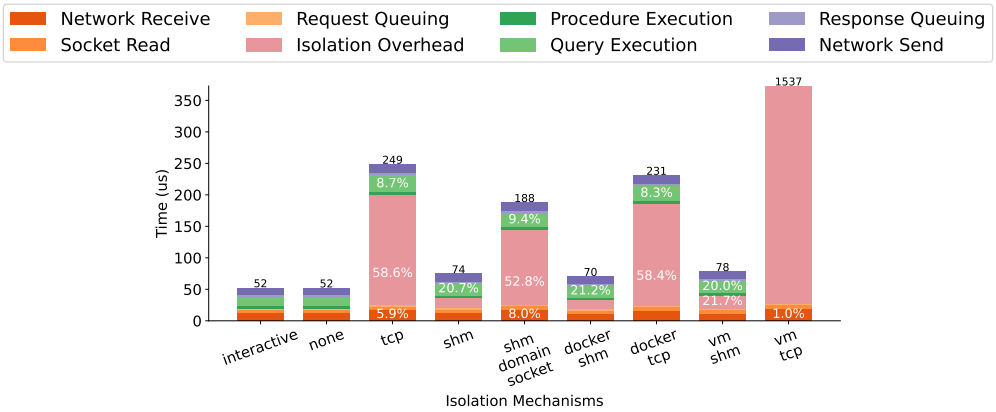
Figure 2.6: **Median Latency vs. Load Across Various Isolation Mechanisms**

**Latency vs. Throughput.** We next explore the end-to-end latency-throughput trade-offs for different isolation mechanisms and examine server-side overhead in detail. The results for different isolation mechanisms are shown in Figure 2.6. Generally, all configurations experience latency increases as more load is applied to them and systems get more saturated. Not surprisingly, the configuration with no isolation (labeled `none`) has the best latency-throughput trade-off curve, as there is no IPC overhead between the OLTP engine and the stored procedure. It achieves up to 65% higher throughput compared to the next-best-performing ones when the highest load is present. The next best-performing configurations are the ones with shared memory and polling (`shm`, `docker_shm`, `vm_shm`). At low load, the latency is very close to the baseline with no isolation. With more complex transactions and higher load, shared-memory adds a couple of milliseconds of latency and tops out at dramatically less throughput. This is mostly due to serialization and de-serialization overhead. Configurations with TCP and domain sockets have much worse latency and achieve significantly lower throughput compared to shared-memory and no-isolation. Domain sockets perform slightly better than TCP because domain sockets in Linux are more efficient than the general TCP/IP stack. We do not observe a significant difference between the procedure process running in a container and running in the host, suggesting the Linux kernel uses a similar code path for the IPC mechanisms. We also observe similar trends for 99-percentile latency for all the configurations.

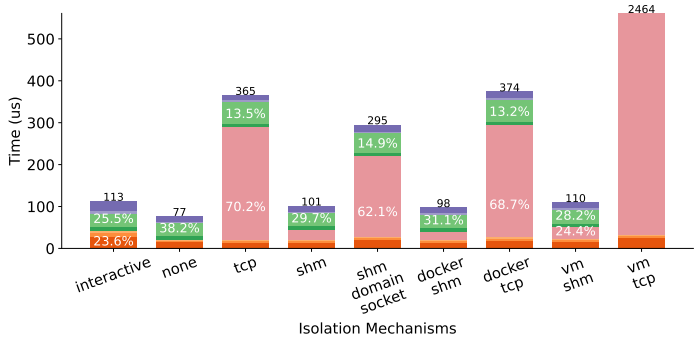
To summarize, isolation imposes dramatic throughput and latency overheads. The best-

performing mechanism involves process isolation with shared-memory and polling. Containerization does not impose significant overhead on top of process isolation.

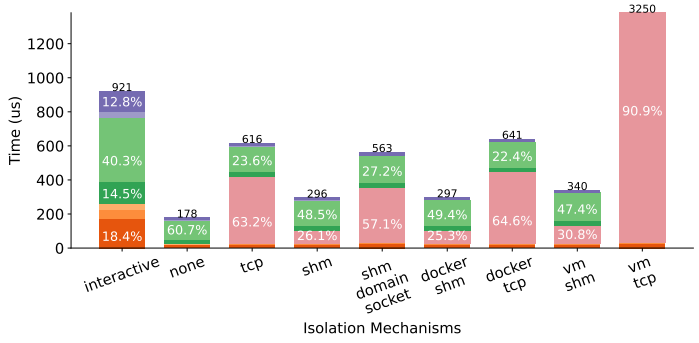
**Server-side Transaction Latency Breakdown (excluding queuing).** To understand the performance of the SP model, we categorize server-side latency of a transaction into the 8 phases highlighted using colored numbers in Table 2.3.




(a) **YCSB**




(b) **Voter**




(c) **TPC-C**

Figure 2.7: **Server-side Transaction Latency Breakdown (Excluding Queuing)**

**Server-side Transaction Latency Breakdown.** As before, we perform the same server-side transaction latency breakdown. The results are shown in Figure 2.7, which provides a detailed analysis of the three benchmarks: YCSB, Voter, and TPC-C using different isolation mechanisms.

One should note that all mechanisms that provide some degree of isolation are primarily dominated by the CPU path length in the communication stack. Specifically, using process isolation and TCP, `Isolation Overhead` dominates server-side overhead, taking up 58%/70%/63% of the CPU time on YCSB/Voter/TPC-C workloads. Each DB interaction requires a traversal through the local TCP/IP stack between the procedure process and the OLTP engine. Unix domain sockets help reduce the overall overhead by about 32% compared to TCP, as the kernel path is shorter. However, it is still significantly higher than no-isolation. The next best-performing configuration is process isolation with shared memory and polling for communication that still has 34%/31%/66% higher overall overhead compared to no isolation. Note that with shared memory and polling, we do not observe a significant difference between running the procedure process in the host and container. This is because shared memory allows for exchanging messages directly through physical memory, bypassing the container boundary completely.

Note that, when running the procedure process in a virtual machine, `Isolation Overhead` drastically increases. To isolate why virtualization introduces such high overhead, we ran a micro-benchmark for TCP using `sockperf`, obtaining the following round trip times (RTT):

| **Config** | **RTT** |
|---|---|
| VM ↔ VM | 60us |
| VM Guest ↔ Host | 176us |
| TCP via Loopback | 21us |

Surprisingly, the RTT between two Google Cloud instances is lower than that of the RTT between a guest VM and host on the same instance. From the table, we can see that the overhead introduced by the guest VM is about 8.5× compared to communicating through the loopback interface on the same host. This is consistent with the difference between `vm tcp` and `tcp` shown in Figure 2.7a. We attribute this high cost to QEMU/KVM and nested virtualization, as each packet needs to traverse two TCP/IP stacks and the VM boundary in one direction.

## 2.4 Lessons Learned

We summarize the key lessons from the experimental evaluation below.

**Lesson 1.** Both the interactive transaction model and stored-procedure model are bottlenecked on messaging when transactions are simple (YCSB/Voter), accounting for 52%–67% of the runtime overhead. The stored procedure model helps relieve these bottlenecks when transactions are more complex at the expense of security. Kernel-bypass and user-space network stacks can help reduce such overhead down to 28%–46%.

**Lesson 2.** On modern networks, the stored-procedure model still has substantial latency and throughput advantages over the interactive transaction model. These increase with the number of interactions with the DBMS. The two models are on par when a transaction contains only one round trip with the DBMS.

**Lesson 3.** Stronger isolation mechanisms for stored-procedures introduce substantial overhead. For example, process isolation with shared memory and polling offers a good balance but still incurs 31%–66% higher overhead than no-isolation. TCP/IP and domain socket communication

for process isolation further increase that overhead. Lastly, virtualization imposes an order of magnitude higher overhead on communication.

Together, these lessons identify communication and isolation as the two DB–OS interfaces where the gap between database needs and OS-provided mechanisms is largest.

## 2.5 Implications for Practical DB-OS Co-Design

These measurements show that modern database engines have made enough progress on internal execution that the next large gains now come from the boundary between the DBMS and the operating system. The most consequential interfaces at that boundary are networking, memory management, and isolation.

### 2.5.1 Communication Path and Practical Networking Co-Design

The most direct implication of the measurements is that the communication path needs a database-aware redesign. Even in the stored-procedure model, where the number of client/server round trips is reduced, kernel network-stack processing, socket processing, and task dispatching consume a large fraction of CPU time. This finding is not limited to VoltDB's architecture. PostgreSQL also spends substantial CPU time in networking when data fits in memory, and cloud-disaggregated OLTP systems make the communication path even more important because log persistence and buffer-pool misses involve network I/O to remote storage servers [43,44].

We also find that DBMSs do not share a single network-I/O architecture. Some systems execute transaction logic and network processing in the same thread, while others dedicate separate threads to network I/O. Table 2.4 summarizes this design split. This lack of consensus is important: a useful DBMS networking stack cannot assume one execution model, and it must support both compatibility with existing network-thread architectures and tighter co-design when the DBMS can expose more structure.

| Unified TX & Network Thread | Dedicated Network Threads |
|---|---|
| PostgreSQL | Redis |
| MySQL | VoltDB |
| SQL Server | Oracle |
| AWS Aurora | Cassandra |
| TiDB | ScyllaDB |
| | OceanBase |

Table 2.4: **How DBMSs Perform Network I/O.**

Existing kernel-bypass mechanisms point in the right direction but are not sufficient for production DBMSs. DPDK can reduce kernel overhead, but it requires exclusive NIC access, busy polling, and substantial rewrites of the DBMS networking layer. It also removes standard Linux networking tools from the fast path, complicating routing, debugging, deployment, and monitoring [45,46]. To our knowledge, Yellowbrick [36] is the only DBMS that makes substantial

production use of DPDK-style kernel bypass. Chapter 3 addresses this gap by building a database-aware networking stack that recovers kernel-bypass-class performance while preserving kernel-driver compatibility and avoiding exclusive device ownership.

The same usability issue appears on the client side. Application servers and API servers also communicate with the DBMS through kernel network stacks, so end-to-end latency and deployment cost depend on both sides of the connection. Traditional DPDK-style deployments are even less attractive on such machines because they are often multi-tenant and cannot dedicate a NIC exclusively to one application. A practical networking mechanism therefore needs to work not only on controlled database servers, but also in environments where normal Linux networking, routing, monitoring, and device sharing must remain available.

User/kernel bypass mechanisms based on eBPF suggest another path. Recent work has shown that database proxying [46], caching [47], and even larger database components [48] can be pushed closer to the OS networking stack. However, eBPF's safety model restricts the programming model: loops must be bounded, kernel data structures are constrained, and floating-point operations are unavailable. These restrictions make it difficult to move complex DBMS logic directly into the kernel. The implication for this thesis is that the right interface should preserve the deployment advantages of kernel-resident hooks while giving the DBMS a richer programming model for its own fast path.

### 2.5.2 Isolation and Privilege: Beyond User-Space Mechanisms

The isolation results expose a different DB–OS boundary. Stored procedures reduce communication overhead, but stronger isolation mechanisms introduce substantial cost. Process-level isolation provides practical safety guarantees through virtual memory and privilege separation, yet the DBMS must communicate across expensive OS boundaries to use it. Shared-memory polling can reduce latency but spends CPU aggressively; sockets and domain sockets are easier to deploy but add communication overhead; full virtualization is substantially more expensive. The key implication is that existing mechanisms do not simultaneously satisfy safety, low communication overhead, and high efficiency. This makes isolation not only a security concern, but also a performance-interface problem between DBMS and OS. A promising direction is to build better IPC and scheduling mechanisms around process-level isolation, rather than treating isolation as a fixed cost outside the DBMS design space.

These goals require tighter control over hardware mechanisms that are currently hidden behind kernel interfaces, including interrupt delivery, virtual memory, and privilege separation. In a conventional user-space process, accessing these mechanisms requires traversing kernel paths whose semantics are not tailored to DBMS execution. Chapter 5 pursues this implication by using hardware virtualization to give one database process safe access to privileged hardware while leaving the host kernel and POSIX ecosystem intact.

### 2.5.3 Memory Management: Beyond OLTP Communication

This chapter focuses on OLTP communication and isolation, but it leaves open an important question for other workload classes. For analytical and vector search workloads where the memory management and storage stack is exercised more heavily, the dominant costs may lie elsewhere. We therefore carry out additional measurements in Chapter 4 to identify where CPU time goes for

buffer management and virtual memory for data-intensive workloads such as OLTP and vector search.

## 2.6 Summary

This chapter revisited the classic OLTP bottleneck analysis on modern hardware and modern systems. We found that the dominant cost has moved to communication and isolation at the DB–OS boundary. Messaging accounts for majority of CPU cycles in simple transactions, and stronger isolation for user-defined logic compounds the cost. Existing remedies—kernel-bypass stacks for networking—are effective in isolation but impose deployability, security, or engineering costs that have prevented production adoption. The following chapters address these bottlenecks in turn, building database-oriented mechanisms at the DB–OS interface while preserving compatibility and deployability.

# Chapter 3

# Practical and Efficient Networking for Database Systems

## 3.1 Introduction

The previous chapter showed that communication is the dominant bottleneck in modern memory-optimized OLTP systems, with much of the overhead coming from kernel network-stack processing, socket processing, and task dispatching. The goal of this chapter is to remove this overhead from the DBMS communication path while retaining the properties that make kernel TCP attractive in production: reliability, compatibility with existing DBMS architectures, and standard deployment and debugging tools.

Existing kernel-bypass systems target kernel stack overheads by moving packet processing into user space, typically with user-space NIC drivers such as Data Plane Development Kit [13] (DPDK) and user-space TCP stacks [14–17,49]. Some stacks [16,17,49,50] go further by running application logic directly on the network core responsible for packet processing, thereby avoiding OS scheduler involvement on the request path. However, these approaches leave several DBMS-specific challenges unaddressed:

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

**Integration and Compatibility Challenges.** User-space network stacks [16,17,49,50] often require substantial architectural changes, which limits their adoption in complex database systems. These stacks typically assume a restricted run-to-completion execution model, where application logic executes directly on networking cores to eliminate scheduler and inter-thread communication overheads. This model works well for simple key-value caches like Memcached, where responses are generated immediately for each request, but it does not generalize to DBMS workloads that

may involve query execution, storage-engine interaction, distributed transactions, or replication. Moreover, many production DBMSes [29,51,52] use dedicated I/O threads that dispatch requests to worker pools for pipelining. Forcing these systems to run entirely on networking cores would require major rewrites. Deployment is another barrier. User-space drivers such as DPDK require exclusive control of a dedicated NIC, preventing coexistence with standard kernel networking tools and complicating debugging, monitoring, and cluster management. To the best of our knowledge, only ScyllaDB [53,54] and Yellowbrick [55] support kernel-bypass networking, and ScyllaDB does not deploy it in production due to deployment challenges [35].

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

This chapter shows that it is possible to remove the OS-kernel stack bottlenecks while addressing these challenges. Our solution is Tux, which moves the network stack to user space, provides flexible interfaces for co-designing network processing with DBMS execution, exploits reliable message-based communication to avoid byte-stream overheads, avoids kernel context-switch overhead when dispatching tasks, and judiciously leverages modern Linux I/O interfaces for compatibility.

At its core, Tux utilizes a kernel-bypass data path with a reliable transport protocol designed for flexibility and performance. It decouples reliability from strict ordering, offering both a TCP-compatible byte-stream interface for zero-change integration and a native message-based interface to eliminate framing and copy overheads. To tackle context switching overheads, Tux introduces a pushdown abstraction that extends the eBPF paradigm into userspace, allowing DBMS logic (e.g., storage look-ups, transaction processing, command processing) to run directly on networking cores inside the Tux stack in the form of callback functions when transport-level data units (stream or message) arrive. Pushdown abstraction can further leverage the message interface for zero-copy data access within callback functions, avoiding overheads. This abstraction enables a wide range of scheduling flexibility to avoid scheduling overhead and data copying. For example, short transactions can be executed directly at the networking cores without incurring context-switching overhead and data copying, while long transactions can be directed to the DBMS's internal pool of worker threads. Furthermore, Tux runs callback functions in a preemptible user-space task runtime that prevents callback functions from monopolizing CPU resources. For compatibility, Tux leverages well-maintained **kernel-space drivers** to communicate with NIC hardware. This is accomplished via a recent kernel-based bypass path using eBPF/XDP, which avoids the NIC exclusivity requirement. Although using kernel-space drivers inevitably introduces system call (kernel-crossing) and interrupt overhead, we find that the CPU cycles spent in kernel NIC drivers contribute only a small portion of overall DBMS request processing latency. Therefore, using a kernel-based approach for better compatibility represents a worthwhile trade-off. Additionally, Tux aggressively exploits batching and kernel-side polling to amortize kernel-crossing overhead and avoid interrupt overhead, allowing it to match the performance of using user-space drivers.

We implement our design in a library called Libtux that provides two operating modes: compatibility mode and pushdown mode. In compatibility mode, Libtux operates as a shim layer with POSIX interfaces for existing DBMSes that rely on TCP, providing binary compatibility and

no DBMS code change. In pushdown mode, LibTux provides an event-driven programming model to minimize porting efforts and facilitate zero-copy messaging.

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

The remainder of this chapter describes the design and implementation choices behind these results. It first presents Tux's hybrid kernel-bypass stack, which uses eBPF/XDP to retain compatibility with kernel-space NIC drivers while moving the performance-critical data path to user space. It then develops the reliable message-oriented data path and pushdown abstraction that let DBMS logic run directly on networking cores when doing so is profitable. The evaluation studies Tux across five DBMSes and shows that these benefits require minimal changes to the DBMS engine.

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

Table 3.1: **Transport protocols used in DBMSes and whether dedicated I/O threads are employed.**

## 3.2 Background and Motivation

This section focuses on the background most directly needed to motivate Tux, including DBMS networking requirements, modern data path I/O interfaces (DPDK, io_uring, eBPF/XDP/AF_XDP), and kernel-bypass designs. We explain why existing solutions are insufficient for our setting and leave broader, more distantly related work to Section 3.6.

### 3.2.1 DBMS Expectations for Networking

We begin by surveying the transport protocols used in major database systems to understand their core requirements, which are summarized in Table 3.1. Our findings show that communication largely falls into two categories—client-to-server and internal inter-node traffic—each with distinct needs.

**Client-DBMS Communication.** For client-server traffic, mainstream database systems rely on TCP. This is a direct consequence of their wire protocols, which are typically connection-oriented for tasks like authentication and expect the strict ordering of a byte-stream to correctly

pipeline requests and responses [56–58]. TCP's wide support and its reliable, ordered byte-stream interface make it a natural and ubiquitous fit for this role.

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

This analysis highlights that modern database systems simultaneously require both an ordered byte-stream interface, and an efficient, reliable message-passing interface. This dual need poses a challenge for existing transport protocols, which are often too general-purpose or require specialized hardware. Homa [59,60], for example, is a recent message-based transport protocol designed for datacenter RPC workloads but lacks a byte-stream interface. Scalable Reliable Datagram (SRD) [61,62] and RDMA [63] also employ message-based protocols but are tied to proprietary hardware or specific network fabrics. Hence, one of the design goals of Tux is to provide efficient messaging over Ethernet that supports both reliable unordered messaging and ordered byte-stream for easy integration into database systems.

### 3.2.2 The Cost of Task Dispatching and Kernel Bypass Approaches

Task dispatching—notifying worker threads of packet arrivals—creates significant overhead in high-performance DBMSes. Packets need to traverse hardware interrupts, kernel TCP stack, and OS scheduler wakeups; dedicated I/O threads (Table 3.1) add further IPC costs. Our microbenchmark (Figure 3.1) shows kernel-mediated IPC incurs multi-microsecond latency, while busy-polling is too CPU-intensive for multi-threaded OLTP systems. Existing kernel bypass systems use two strategies: co-located [16,17,49,50] (networking and application on same core, eliminating IPC but vulnerable to head-of-line blocking) or separated [14,15,64–66] (flexible scheduling but higher communication overhead).

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

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

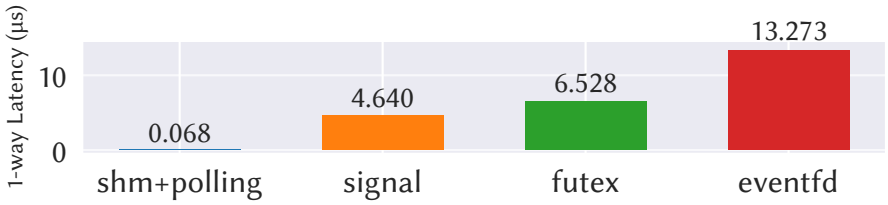



Figure 3.1: **Latency of IPC Mechanisms** - The benchmark measures the average one-way latency of ping-ponging $10^6$ 1-byte messages between two threads on a security-hardened [67] EC2 instance

### 3.2.3 Modern Data-Path Interfaces

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

**User-Space Driver (DPDK).** DPDK [13] lets an application take complete control of a NIC and process packets in user space, bypassing the kernel TCP stack. The application runs a loop that busy-polls a receive queue and then process the packets (e.g., hands the packet to a user-space TCP stack such as F-stack), and then transmits packets—eliminating interrupts and scheduler wakeups. This yields very low per-packet overhead and predictable latency, at the cost of CPU spent on polling and exclusive NIC ownership.

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

**eBPF.** The extended Berkeley Packet Filter (eBPF) is a modern Linux technology enabling safe and efficient kernel extensions without modifying the kernel or loading kernel modules. By running sandboxed programs at various kernel hook points—including the networking stack—eBPF allows integration of data-intensive operations directly into the OS networking stack. Recent research has demonstrated its potential for tasks such as database proxying [46], caching [47], and transaction processing [68,69], effectively avoiding costly data transfers between user and kernel space. However, eBPF enforces strict safety guarantees: loops must be bounded, programs are limited to accessing a restricted set of kernel data structures, and floating-point operations are prohibited in the kernel, making it challenging to offload a full-fledged DBMS into kernel space.

**Kernel-Space Driver through AF_XDP sockets.** Building on eBPF, the eXpress Data Path (XDP) [70] provides a hook that runs eBPF programs as soon as a packet arrives at the network card's driver. From here, an XDP program can decide the packet's fate: drop it, pass it to the normal kernel stack, or—most importantly for Tux —redirect it to a user-space application through a high-speed "express lane" called an **AF_XDP socket**. These sockets use a shared memory region that both the application and the NIC can access directly. This enables true zero-copy packet processing, as data copying between kernel and user space can be elided. Tux leverages this mechanism to bypass the kernel stack and quickly get packets to its user-space data path.

### 3.2.4 The Cost of Using Kernel-space NIC Drivers

While using kernel-space NIC drivers offers superior compatibility, it does introduce kernel-crossing overhead for NIC operations. To quantify this overhead, we perform a micro-benchmark by running Redis on F-Stack [16] atop DPDK—which employs a run-to-completion execution model—providing an upper bound on achievable performance. The following table breaks down the average latency (excluding polling/queuing time) of Redis serving GET requests at a throughput of 300K/s:

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

The raw NIC driver accounts for only 7.2% of the total time. A kernel-space driver adds the cost of a system call. On a security-hardened [67] `c5n.xlarge` AWS instance, a minimal system call takes about 390ns. This would add roughly 390ns for receiving and 185ns for transmitting, potentially increasing the driver-related portion of latency to around 20% of the total for this simple Redis workload. However, this analysis is conservative. For a more complex database operation taking 20µs, this same system call overhead would drop to just 2.6% of the total time. This reasonable performance trade-off motivates Tux to use kernel-space NIC drivers, gaining compatibility and ease of deployment for a modest cost.

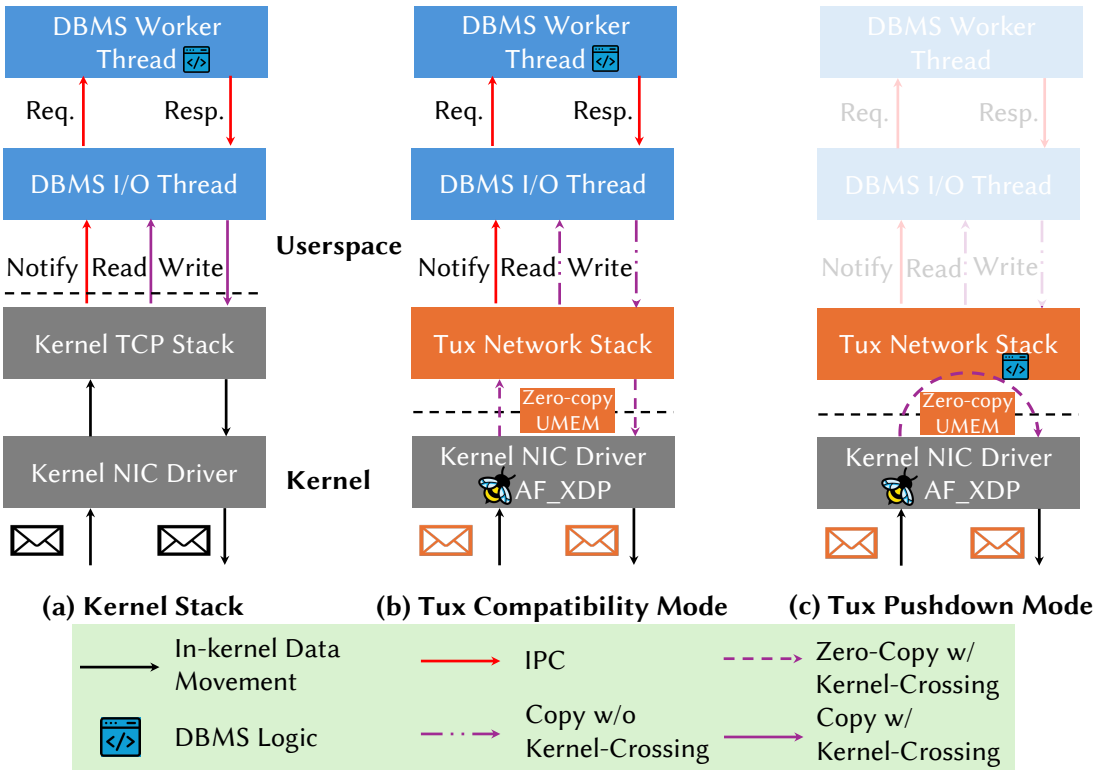



Figure 3.2: **How Requests and Responses Enter and Leave a DBMS Worker Thread** - (a) Kernel Stack (b) Tux Compatibility Mode speeds up the network stack with specialized messaging protocol and efficient compatible kernel-based bypass path. (c) Tux Pushdown Mode speeds up network stack and avoids IPC overhead by pushing performance-critical DBMS logic onto networking cores without requiring an overhaul to DBMS's architecture.

## 3.3 Tux Design

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

- **G1: Preserve compatibility & high performance.** Tux must support unmodified DBMSes while avoiding the need for kernel modules or exclusive NIC access. At the same time, it must deliver performance comparable to specialized DPDK-based solutions.
- **G2: Enable even better performance via co-design and programmable network stack.** Tux stack should expose lightweight and flexible programming interfaces for DBMS to exploit further performance gains at modest engineering effort.

**Overview.** These goals shape the overall design of Tux, shown in Figure 3.2 (b) and Figure 3.2 (c). To achieve **G1**, Tux runs as a user-space library with POSIX interfaces under the DBMS. When a kernel TCP connection is established, Tux transparently replaces it with a user-space data path that bypasses the kernel TCP/IP and socket layers, without requiring application changes or NIC

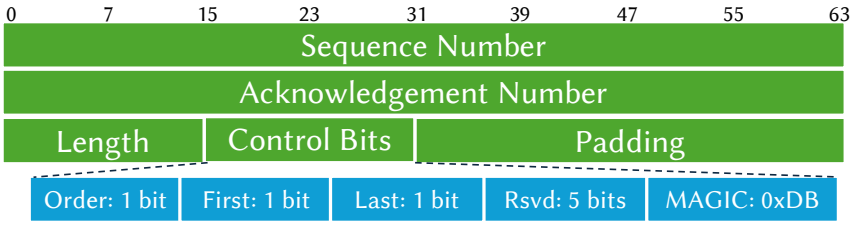



Figure 3.3: **Tux Packet Header**

ownership. The data path uses the reliable message-oriented Tux protocol (Section 3.3.1), which exposes a native message interface and can emulate byte-stream delivery for compatibility. The Tux protocol is implemented in Tux stack (Section 3.3.2) on top of recent kernel-based bypass path using eBPF/XDP for accessing NIC driver. Tux stack includes several optimizations (Section 3.4) to minimize kernel crossing for using the kernel space NIC drivers.

**Pushdown.** To achieve **G2**, Tux introduces pushdown (Section 3.3.3): a programming abstraction that lets DBMS logic run directly on networking cores in the form of callbacks, avoiding kernel scheduler and IPC overhead. Pushdown elevates the unit of execution from raw packets to transport-level messages or byte-streams, so DBMS developers can implement request handling at the same granularity as DBMS engine. Conceptually, pushdown extends the eBPF/XDP paradigm into user space: it provides programmable hooks on the data path, but with the full flexibility of user space rather than kernel restrictions. Internally, pushdown is realized via a two-stage pipeline: an in-kernel eBPF/XDP program that quickly steers database traffic through a zero-copy packet buffer to user space, and then Tux executes callbacks inside a preemptible runtime that ensures robustness and fairness, even for long-running logic. Furthermore, callback can tap into the native message interface of Tux protocol to avoid copying and parsing.

### 3.3.1 Tux Data Path Protocol

The Tux data path uses a reliable specialized transport mechanism designed specifically for database workloads within the Tux stack. A key design principle is the decoupling of reliability from strict in-order delivery, inspired by SCTP protocol [71]. Unlike stream-based TCP, Tux inherently operates on discrete messages, aiming to eliminate application-level framing and parsing overhead.

**Protocol Mechanics:** The structure of a Tux packet, shown in Figure 3.3, serves the following purposes:

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

fragmented message, simplifying reassembly at the receiver.

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

**Message Semantics.** For communication patterns where message boundaries are essential but strict ordering is not, Tux provides native message semantics. The sender transmits packets with the `order` bit cleared. The receiver buffers incoming packets, reassembling a complete message (using `first`/`last` bits if fragmented) before delivering it. Delivery can occur via POSIX `recvmsg` or, in pushdown mode (Section 3.3.3), directly to a callback.

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

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

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

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

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

**Comparison to SCTP.** While both Tux and SCTP offer reliable, message-oriented transport beyond TCP, Tux is not general-purpose and is specialized for database systems for both compatibility and performance. SCTP is a general-purpose message protocol mainly used in mobile networks with features [72] like multi-homing, multi-streaming, and a complex, extensible chunk-based packet format that are unnecessary in database workloads. In contrast, Tux uses a simpler header structure tailored for its dual stream/message interfaces and relies on UDP encapsulation for common checksum hardware offloads, which are less frequently available for SCTP. Thus, Tux prioritizes performance and tighter integration with database systems.

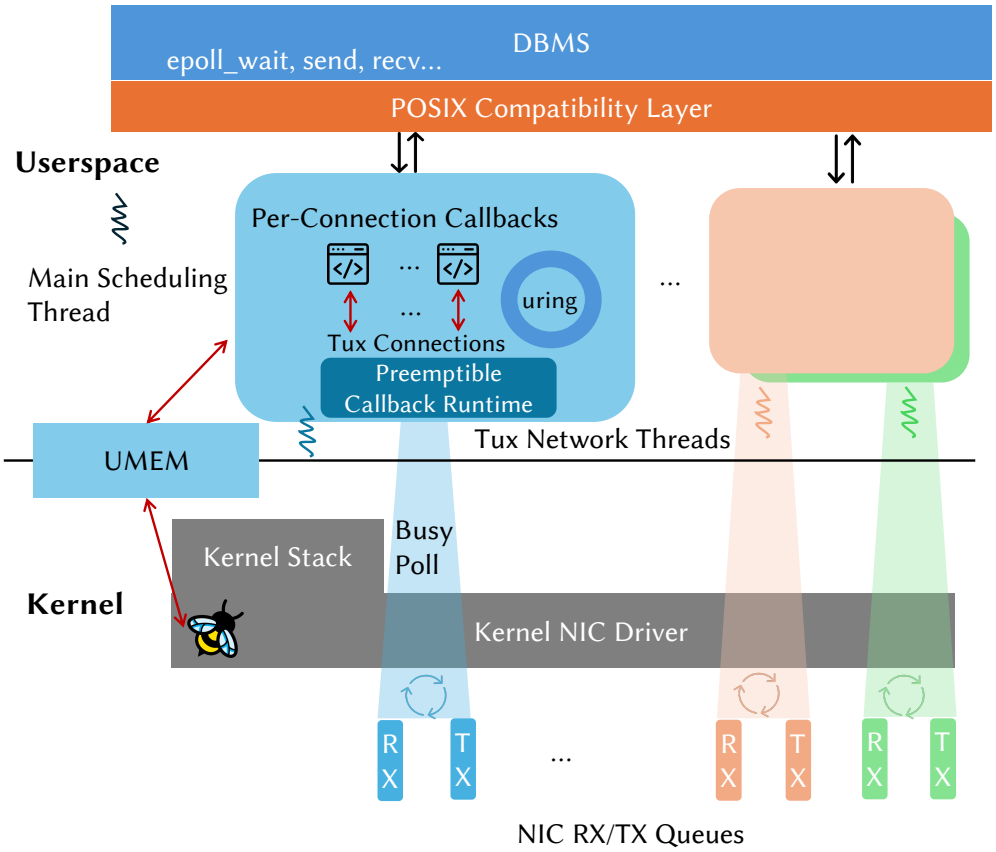



Figure 3.4: **Tux Stack** - Each Tux network thread manages a set of NIC queues via a corresponding set of AF_XDP sockets. Due to NIC receive-side scaling, packets of a Tux connection will also be mapped to the same NIC queue and, hence, the same Tux network thread.

### 3.3.2 Tux Components

Figure 3.4 presents the internals of Tux stack which consists of a compatibility layer, a shared-nothing architecture for managing concurrency and memory, the Tux runtime for running pushed-down callback functions, and kernel-space NIC drivers, as discussed below.

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

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

compatibility.

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

**Memory Management**. To eliminate contention, Tux allocates a user-space memory region per NIC-queue shared between userspace and kernel via the AF_XDP socket APIs [73]. The region is divided into 4KB pages. Each page may be used as a packet buffer for both receiving and transmitting by the NIC hardware. This ensures that all packets belonging to a Tux connection reside in a memory region (UMEM region) shared between userspace and kernel, avoiding coordination between NIC queues.

### 3.3.3 Pushdown Abstraction and Runtime

The pushdown abstraction can be viewed as a user-space extension of the eBPF paradigm, but operating at the transport layer rather than on raw packets. Whereas kernel eBPF provides a safe but restricted environment for filtering and simple packet operations, Tux pushdown exposes a programmable interface on DBMS-relevant transport units: reliable messages and ordered byte streams. This design elevates the programming model from low-level packets to semantically meaningful objects—such as a SQL request, a log page, or a key–value lookup—matching the abstractions DBMS developers actually care about. Unlike kernel eBPF, pushdown executes unrestricted user-space logic with access to the full runtime environment, while the Tux stack manages transport-level complexity such as reliability, fragmentation, and reassembly. This approach combines the efficiency of kernel-bypass with programmability, enabling DBMS developers to attach logic directly to transport events. Currently, Tux supports two types of per-connection callbacks on arrival of Tux messages:

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

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

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

Figure 3.5: **Tux Preemption Signal Handler** - Preemption is skipped when a tux routine is in an uninterruptible state

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

**Tux Lightweight Task.** A Tux callback is executed inside a lightweight user-space thread, called Tux routine, that runs on the networking core, similar to a go-routine [74]. Each routine maintains its own stack and a small region of memory for storing register values during context switches on preemption. We avoid stackless user-level threads [75] as a stack-based design provides better compatibility with unmodified user code, despite slightly higher context switch overhead [76].

**Grouped Execution.** Running each Tux callback function in its own routine is inefficient, as it incurs one context switch per invocation and requires a stack per routine. To reduce both overheads, Tux runs multiple callbacks in a single Tux routine. This amortizes the cost of switching and reduces memory usage. When a new group of callbacks is ready and there is no idle or preempted routine available, Tux creates a new routine to run them. As a result, the number of active routines is proportional to the number of preempted executions, not the total number of Tux connections.

**Scheduling.** Tux time-shares between callback execution and network processing. Callbacks are assigned a fixed time slice, while network processing runs without time constraints. Within the time slice, Tux schedules callbacks using FIFO with a timeout. As shown in Figure 3.6, callbacks are invoked in arrival order from a task queue. If the time slice expires while a callback is still executing, the routine is preempted and moved to the end of the queue, allowing other callbacks to run in the next round. This prevents a long-running callback function from monopolizing CPU resources. To handle blocking I/O, Tux converts common `glibc` calls (e.g., `pread/pwrite`) into asynchronous I/O via `io_uring`, voluntarily yielding control back to the Tux scheduler.

**Preemption.** Tux uses Linux signals to perform preemption for its wide availability in both cloud VM and bare-metal hardware. The main scheduling thread (Figure 3.4) checks the run time of each active tux routine and sees if it exceeds the allocated time slice. When a violation happens, it sends a signal to the thread running the routine which is processed by a signal handler, as shown

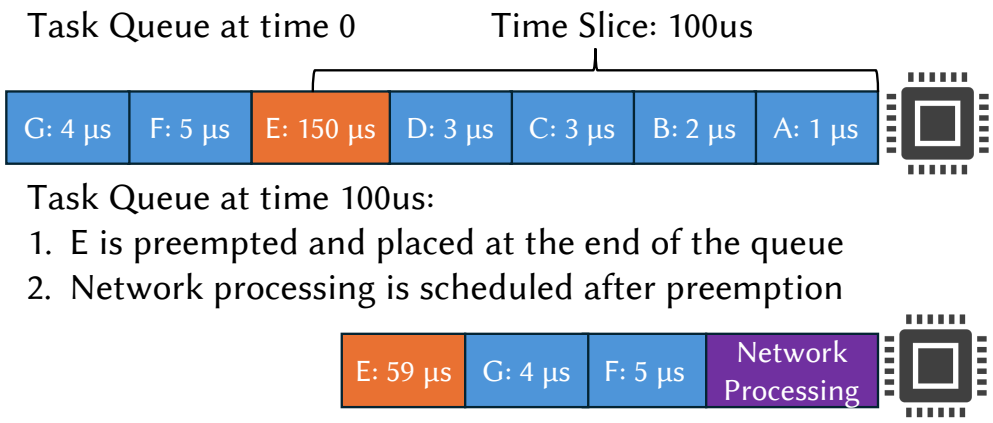



Figure 3.6: **FIFO Scheduling with Timeout** - E is preempted after the time slice is exhausted; E is enqueued to give other tasks a chance to run. Network processing is scheduled between callback executions.

in Figure 3.5. The handler performs a context switch on the actively running Tux routine and switches back to the network processing context. Because the Linux signal has an overhead of a few microseconds, as shown in Figure 3.1, we limit the time slice to be at least 100$\mu$s to make sure the preemption overhead is not significant. This approach offers a practical balance between minimizing tail latency for short tasks and maintaining high overall throughput [77].

**Uninterruptible Region.** There are cases where preemption causes problems. For example, when Tux routine is manipulating Tux runtime data structures, preemption might cause the runtime data structures to be in a corrupted state, resulting in crashes. Tux routine leverages a flag (uninterruptible) in memory to denote whether a callback is in an uninterruptible state. Tux runtime leverages such a flag to build critical sections that cannot be interrupted or preempted. As shown in Figure 3.5, after receiving the preemption signal, Tux checks the flag and aborts the preemption attempt if the uninterruptible flag is set. Tux also makes sure that the preemption handler itself is implemented in a re-entrant fashion.

### 3.3.4 DB-Network Stack Co-Design Patterns

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

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

**Selective Pushdown:** Many database workloads involve a mix of short-running and long-running requests [78]. For more complex transactions (e.g., TPC-C Stock-Level), range queries,

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

Figure 3.7: **Tux Auto Scaling Algorithm**

aggregations, or joins, which might run for longer durations or require coordination/parallelism across multiple cores, it is often beneficial to dispatch them to the DBMS's existing worker pool. Developers can implement this by registering a callback that acts as a dispatcher. The callback parses the incoming request, identifies its type or complexity, and decides whether to process it directly for better efficiency or forward it to a worker pool for better tail latency.

### 3.3.5 Auto-Scaling Tux

Executing callbacks on the network core consumes CPU cycles. When a single core is overloaded, latency will increase dramatically. Tux introduces an auto-scaling mechanism that reacts to CPU overload to spread the load over multiple network threads. Tux adopts a per-core CPU utilization-based scaling strategy. Specifically, Tux tracks the fraction of CPU cycles spent in user space doing useful work, defined as $U_{useful}$, which is derived from $1 - U_{polling}$ where $U_{polling}$ is the fraction of CPU cycles spent in kernel drivers. Tux periodically checks the $U_{useful}$ of each network thread in the main scheduling thread. The scaling steps are described in Figure 3.7. When the $U_{useful}$ estimation exceeds a threshold $T_{scaleup}$, Tux allocates a new network thread and redistributes the NIC queues among the new set of network threads. Due to Tux's shared-nothing design (i.e., a NIC queue and all the connections belonging to the queue are exclusively managed by a single network thread), Tux only supports scaling the number of network threads up to the number of supported NIC queues. Similarly, when $U_{useful}$ drops below a threshold $T_{scaledown}$, Tux removes a network thread and re-balances the NIC queues among the new set of network threads.

### 3.3.6 Discussion

**Deployment.** One limitation Tux has is that it requires both ends to run the Tux stack and protocol. This is reasonable for inter-node communication among a distributed or disaggregated DBMS where DBAs typically have more control over deployment compared to communication between clients and the DBMS. However, we believe this limitation to be less of a concern as eBPF, on which Tux relies, is getting increasingly widespread [79] in the cloud [80]. Compared to DPDK-based solutions, Tux does not require kernel modules or full NIC control or huge-pages. Furthermore, the Tux compatibility mode allows easy integration with applications to connect to a Tux-enabled DBMS server.

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

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

## 3.4 Implementation and Optimizations

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

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

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

**Avoiding Preemption Overhead.** To avoid unnecessary preemption, the Tux runtime tracks the execution time of each routine. If a routine has consumed more than 90% of its time slice

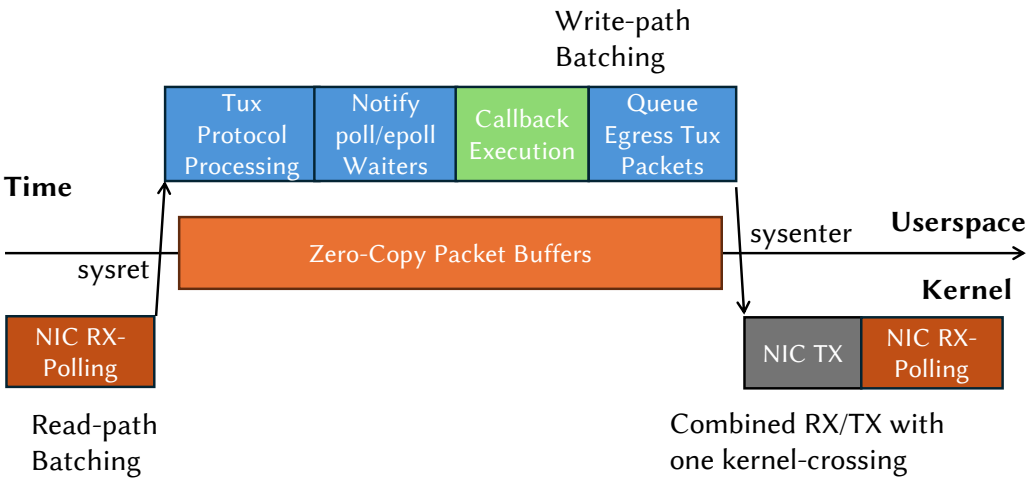



Figure 3.8: **One Iteration of the Processing Loop in a Tux Network Core and Optimizations to Reduce System Call Overhead** - Read-path Batching busy-polls in the kernel NIC driver to collect a batch of packets; Write-path Batching prepares a batch of packets before going into the kernel; Combined RX/TX performs packet transmission and RX polling with a single system call

after finishing a callback, it voluntarily yields control to the scheduler. This optimization allows routines running short callbacks to avoid being preempted entirely, reducing preemption overhead while maintaining fairness.

**Optimizations for using Kernel Drivers.** As explained in Section 3.2.3, using kernel-space drivers introduces kernel-crossing overhead. To mitigate this, Tux's network processing loop, shown in Figure 3.8, is designed to maximize batching and minimize system calls. The loop begins with ***Read-path Batching***. Instead of making a system call per packet, Tux uses a single call to leverage kernel-side polling, which disables interrupts and receives an entire batch of packets into the shared UMEM buffer. After protocol processing, Tux notifies any DBMS threads waiting on `poll/epoll` and then proceeds to execute callbacks, allowing for better pipelining between the network stack and DBMS workers. For ***Write-path Batching***, outbound packets generated by both DBMS threads and callbacks during this process are queued. At the end of the loop, all queued packets are sent in a single batch. This pipeline enables an optimization where the system call to transmit the outbound batch is combined with the call to poll for the next inbound batch.

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

## 3.5 Evaluation

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

- How does Tux perform on real-world high-performance database systems? (Section 3.5.2)
- How much performance benefit does the push-down abstraction provide for database systems? (Section 3.5.3)

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

Table 3.2: **Code changes for porting DBMSes to kernel bypass stacks**

- What are the performance benefits of using native unordered message-based communication for database systems? (Sections 3.5.3 and 3.5.5)
- What are the benefits of tighter co-design between DBMS work scheduling and Tux stack on transaction throughput and tail-latency? (Section 3.5.4)

### 3.5.1 Baselines and Environment

We compare Tux mainly against four kernel-bypass systems: TAS [15] at commit `d3926b`, F-stack [16] at commit `bdd7bed`, and Demikernel [17] at commit `db4b938`, ScyllaDB v6.1.0 with userspace TCP stack running on DPDK v23.07.0. TAS runs a user-space TCP stack in a process as a microkernel service. DBMS connects to the TAS process via shared memory. F-stack and Demikernel are single-threaded user-space TCP stacks that adopts a run-to-completion execution model and runs directly in the same DBMS process. We configure Tux runtime to use a preemption time slice of 100$\mu$s.

All experiments are conducted on two `c5n.4xlarge` instances on AWS. Each instance has 16 vCPUs virtualized from an Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz and 42 GB of memory. The instances are closely placed in a single availability zone and via the *cluster* placement group policy [81]. The system uses an unmodified Linux kernel (v6.8.0) running Ubuntu 24.04. The instances are equipped with Elastic Network Adapter running v2.13.2g driver with 25Gb/s bandwidth. We use a Maximum Transmission Unit (MTU) of 3498 bytes supported by ENA [82]. The measured median round-trip time between two instances is about 34 $\mu$s with DPDK.

### 3.5.2 Comparison on Real-World Systems

We start by evaluating Tux against kernel-bypass baselines on four real-world database systems and a cache system. By default, we use the TAS kernel-bypass stack on the client instance to communicate with the server. This allows a fair comparison as both the server and client sides are running kernel-bypass stacks. By default, we use stream-based callback for Tux pushdown mode as the benchmark focuses on client-DBMS communication, where TCP-style byte-stream compatibility is required. We evaluate the message interface in the next sections. We list code changes required for porting to use various kernel bypass stacks in Table 3.2. F-stack and Demikernel both require orders of magnitude more code changes compared to Tux due to new APIs and architectural constraints.

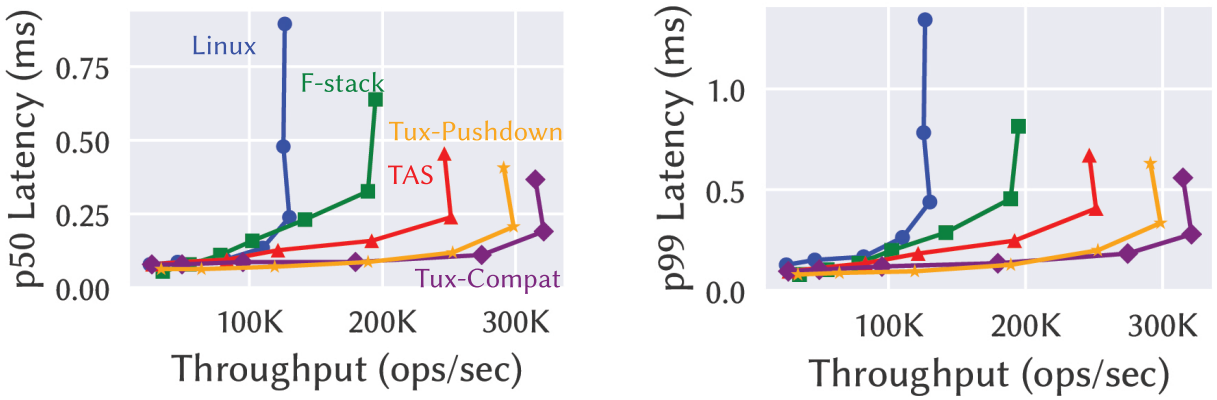



Figure 3.9: **Throughput-Latency on Redis** - Tux variations deliver the highest throughput and lowest latency (Workload: 100% GET, Database: 1M 1KB-sized records).

### Redis

We use `memtier_benchmark` [83] to generate GET requests with a uniform key access pattern to the Redis server storing 1M 1KB-sized records. We increase the load by increasing the number of connections to 128, each with a pipeline depth of 1.

The results are shown in Figure 3.9. Tux-Compat delivers the highest achievable throughput, outperforming Linux/F-stack/TAS by 2.47×/1.64×/1.27×. While Tux-Compat mode achieves higher throughput than Tux-Pushdown, it requires two cores, which pipelines Redis command processing and Tux protocol processing compared to just one core in Tux-Pushdown. When operating at 90% of Linux achievable throughput, Tux-pushdown improves P50 latency by 4.1×/2.63×/1.74×/1.23× and P99 latency by 5.25×/2.45×/1.87×/1.3×, compared to Linux/F-stack/TAS/Tux-Compat, with only 59 lines of modifications to Redis. We omitted results for Demikernel as it does not scale beyond 70K/s throughput due to a bottleneck in its co-routine scheduler.

**Latency Breakdown**. To understand each kernel bypass stack in more detail, Figure 3.10 presents the server-side latency breakdown at a relatively low load (throughput=50K/s). First, all kernel-bypass systems have lower latency compared to the Linux stack, as expected. The reductions are from more efficient network stacks and reduction in kernel-boundary crossings. TAS has a slightly better overall latency compared to Tux-Compat because it uses busy-polling on shared memory which has lower latency than Tux-Compat's epoll-based communication. Tux-Pushdown offers the lowest overall server-side latency for serving a request, despite using kernel drivers, outperforming F-stack by 18%. We attribute this improvement to the optimizations Tux employs for using kernel drivers, efficient implementation, and protocol simplicity.

### VoltDB

We evaluate Tux's compatibility mode on VoltDB (configured with 8 worker threads) against Linux and F-stack. TAS was excluded as it does not support the local loopback TCP connections used internally by VoltDB. Integrating the single-threaded F-stack was invasive, requiring 1300 lines of code change [84] and limiting its evaluation to a single I/O thread. In contrast, Tux works

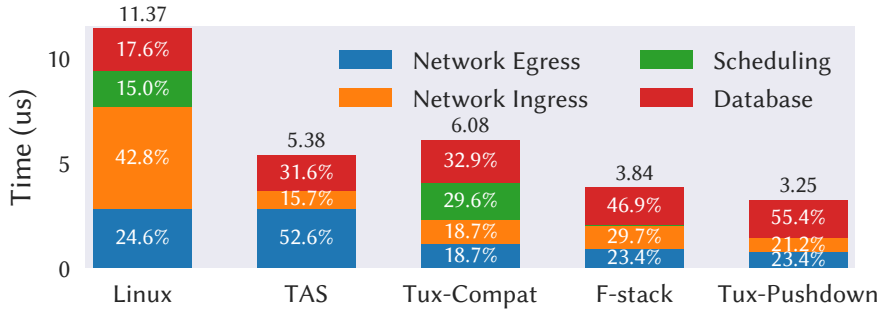
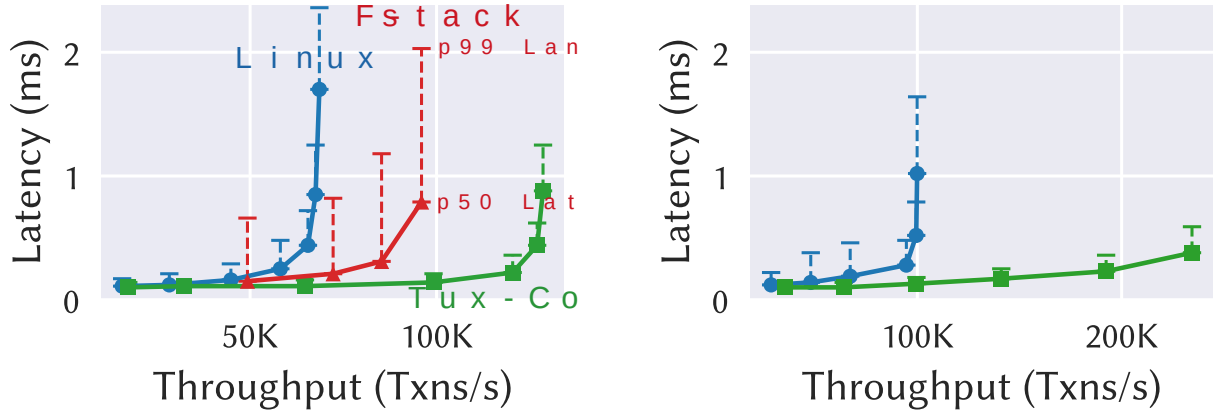



Figure 3.10: **Server-side Latency Breakdown of Redis GET Request on Kernel Bypass Stacks** - (Throughput = 50K ops/s on 1M 1KB-sized records, excluding polling, queueing, and off-CPU time).




(a) **1 VoltDB I/O Thread**

(b) **4 VoltDB I/O Threads**

Figure 3.11: **Throughput-Latency of YCSB-C Transactions on VoltDB** - Marker=p50; whisker=p99. Tux scales to 2.3× higher transaction throughput and significantly improves median and p99 latency. (F-stack results are excluded for 4 I/O threads as it does not support multiple I/O threads for VoltDB)

with zero modifications.

As shown in Figure 3.11, Tux's CPU-efficient protocol and stack deliver significant gains. With a single I/O thread, Tux improves throughput by up to 1.87× and reduces p99 latency by up to 4.72× compared to the baselines. When scaling to 4 I/O threads—a configuration F-stack cannot support—Tux achieves 2.3× higher throughput than Linux, while also reducing median and p99 latency by 2.17× and 2.82× respectively. These results demonstrate that Tux provides substantial throughput and latency improvements on a full-fledged relational OLTP system without any code modification.

### ScyllaDB

We evaluate Tux on ScyllaDB—a system engineered from the outset for maximum performance with an optional user-space TCP/IP stack on DPDK. Its shared-nothing, shard-per-core design [85]

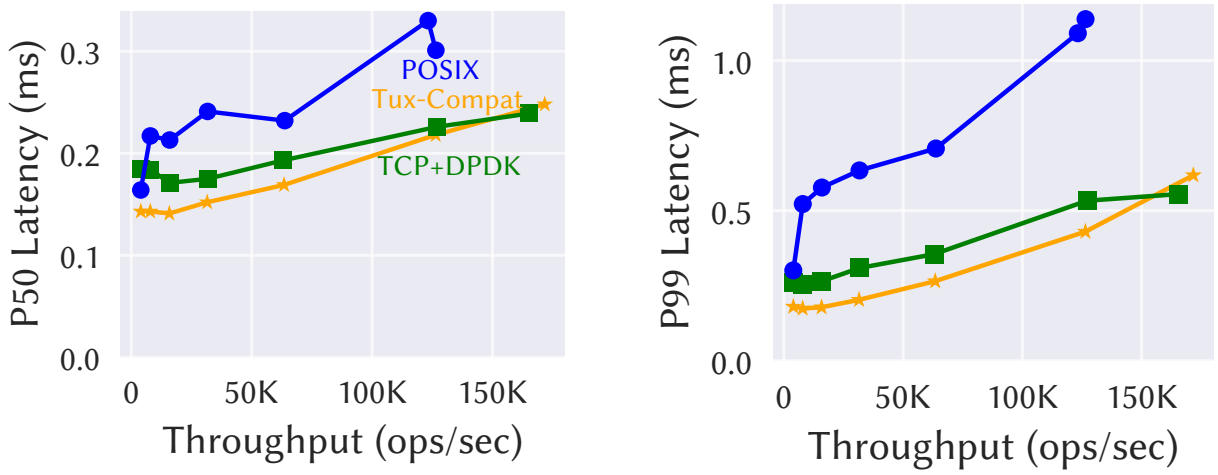



Figure 3.12: **Throughput-Latency on ScyllaDB** - Tux-Compat provides end-to-end substantially lower P99 latency and comparable P50 latency compared to ScyllaDB's DPDK+TCP stack and POSIX stack (Workload: YCSB-C, Database: $10^6$ 1KB records, # Tux Net Thread=1)

intentionally co-locates TCP/IP processing with query execution to minimize kernel involvement and synchronization. Because this design executes each query on a single core without latches, Tux 's pushdown mode (which would run DBMS logic on networking cores) is incompatible: it would require safe cross-core access to database objects. We therefore run Libtux in compatibility mode to transparently replace the POSIX stack. We use `cassandra-stress` [86] to generate a YCSB-C workload, configuring ScyllaDB with 8 CPU cores under both the POSIX and native TCP+DPDK stacks. For Tux-Compat, we dedicate 7 cores to request processing and 1 core to the Tux network thread.

As shown in Figure 3.12, Tux-Compat achieves a similar peak throughput to ScyllaDB's TCP+DPDK stack; both outperform the POSIX stack by 36%. At low to moderate load (<120K ops/s), Tux-Compat delivers 13–23% lower median latency and 20–31% lower P99 latency than the native TCP+DPDK stack. At higher loads, however, the native TCP+DPDK configuration pulls ahead—consistent with ScyllaDB's design intent—because packet processing and request handling run on the same cores, avoiding IPC, and scaling more smoothly. In contrast, Tux 's single network thread becomes a bottleneck as load increases, which is reflected in the steeper latency growth for Tux-Compat. The small gap overall is attributable to Tux 's simple protocol and batching optimizations.

### Memcached

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

The results in Figure 3.13 show that Tux's latency benefits become more prominent as load

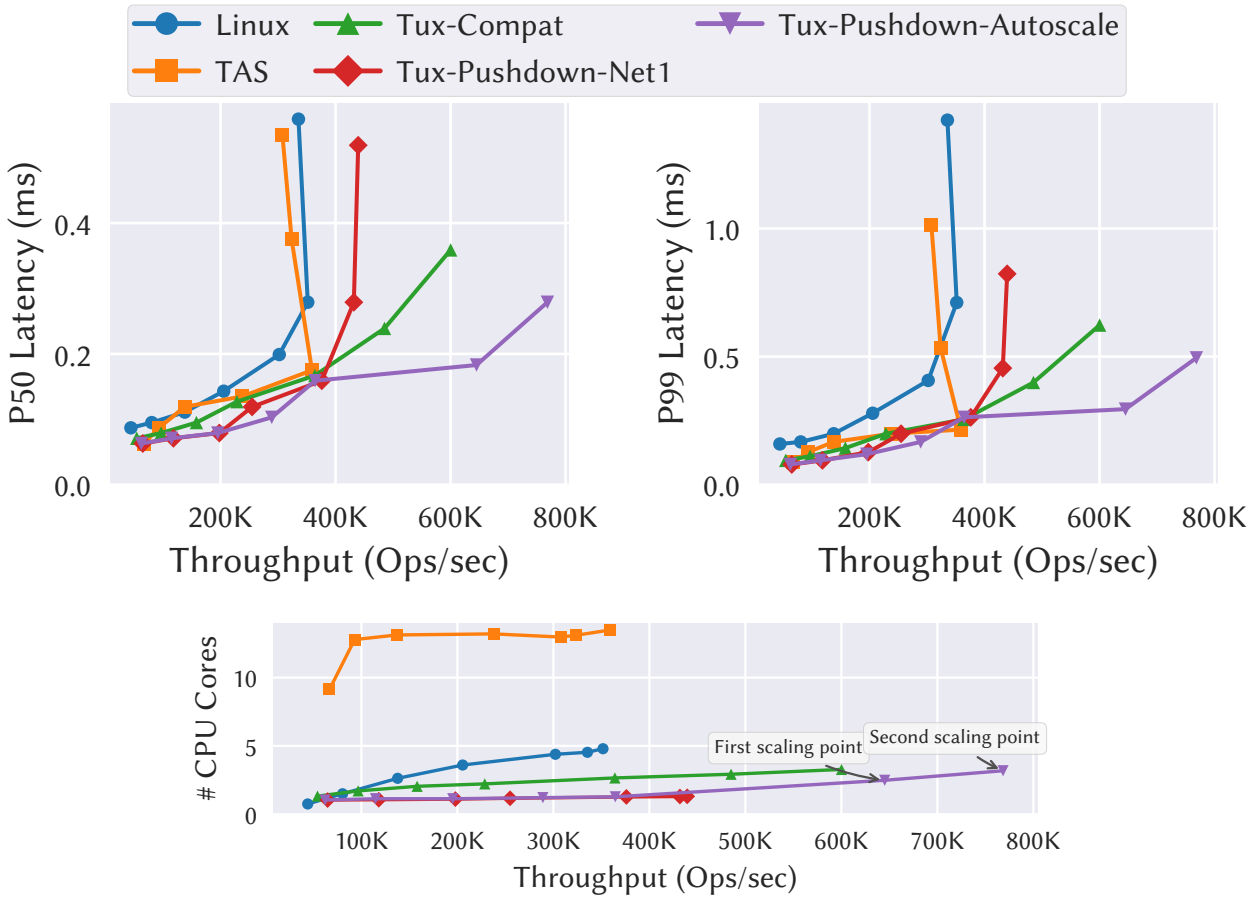



Figure 3.13: **Throughput-Latency on Memcached** - Tux delivers the highest throughput; Pushdown mode consumes the least CPU resources at a given throughput point. (Workload: YCSB-C with 1KB payload; Tux-Pushdown-Autoscale Configuration: $T_{scaledown} = 0.2$, $T_{scaleup} = 0.9$).

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

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

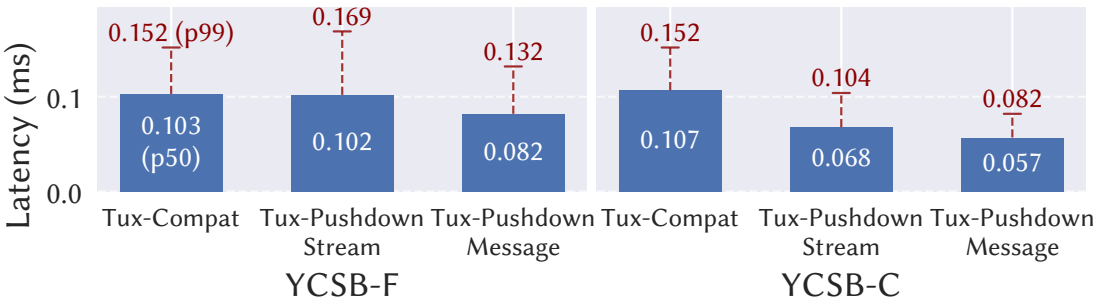



Figure 3.14: **End-to-end Latency of YCSB Transactions on Networked LeanStore** - Push-down results in up to 31/25% reduction on p50/p99 latency; Message interface further reduces the p50/p99 latency by up to 22%/25%. (We use a database of $10^6$ 1 KB-sized records running at 180K txns/second)

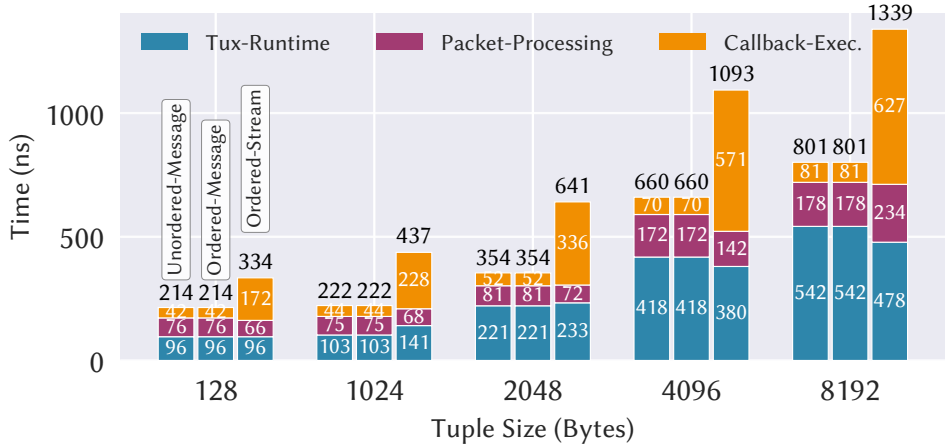



Figure 3.15: **Server-side Per-Request Overhead Breakdown for Various Packet Delivery Approaches** - we categorize CPU time spent in Tux-Runtime (Tux-Runtime), network packet processing (Packet-processing), and executing DBMS logic (Callback-Exec.) .

### 3.5.3 Benefits of Push-down Execution and Message Interface

To quantify the benefits of pushdown execution and the message interface, we built a transactional key-value store server using LeanStore [87] to execute YCSB-C and YCSB-F stored procedures. The server uses a dedicated I/O thread to read from Tux and dispatch requests to a worker thread. In our controlled experiment, we use one I/O thread and one worker thread. The results are shown in Figure 3.14. The Tux-Compat baseline uses three threads (I/O, worker, and network), enabling pipelining, whereas pushdown mode uses only a single thread for all tasks. Despite losing this pipelining, pushdown with a stream callback still reduces median latency on YCSB-C/YCSB-F by 11% and 31% respectively, due to lower scheduler overhead and cache coherence traffic. Switching to the message-based callback provides a further median latency reduction of 22% and 16% by eliminating byte-stream parsing and data copying. Overall, message-based pushdown improves p99 tail latency by up to 70% compared to the pipelined Tux-Compat baseline.

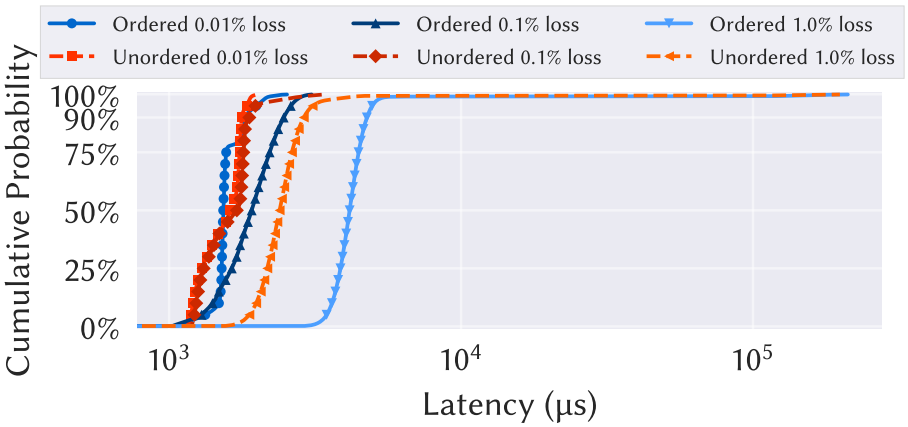



Figure 3.16: **Cumulative One-way Latency Distribution for Transferring 1024-byte Messages under Varying Packet Loss Rate** - Unordered transfer achieves the lowest latency under loss

### 3.5.4 Benefits of DB-Network Stack Co-designs

This section demonstrates how co-designing DB transaction scheduling with Tux optimizes the TPC-C workload on LeanStore, which features a mix of 92% short (Order-Status, New-Order, Payment) and 8% long (Stock-Level and Delivery) transactions. We compare three approaches: (1) **No-Pushdown**, a baseline using Tux's compatibility mode to feed two LeanStore worker threads; (2) **Complete-Pushdown**, which executes all transactions in callbacks on a single Tux network thread; and (3) **Selective-Pushdown**, where a callback acts as a dispatcher, executing short transactions directly but forwarding long ones to the worker threads. For the latter, we evaluate two variants using message groups of size 1 and 32, as larger groups can improve resource utilization via better pipelining.

The results are summarized in Figure 3.17 at a throughput of 31,000 transactions/s. The selective pushdown approach with group size of 32 messages provides the best performance for short transactions, reducing median latency by 30% versus No-Pushdown and 57% versus Complete-Pushdown. The p99 latency improvements are even more pronounced, with reductions of 52% and 69%, respectively. Even long transactions benefit, with an 8% latency reduction compared to No-Pushdown, primarily because the callback bypasses the LeanStore I/O thread, reducing communication overhead.

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

### 3.5.5 Cost of Ordering and Byte-Stream Interface

In this section, we quantify the costs of a streaming interface and of enforcing delivery order. Figure 3.15 shows a breakdown of server-side CPU time when transferring tuples of varying sizes between two machines over a Tux connection. We compare unordered message delivery,

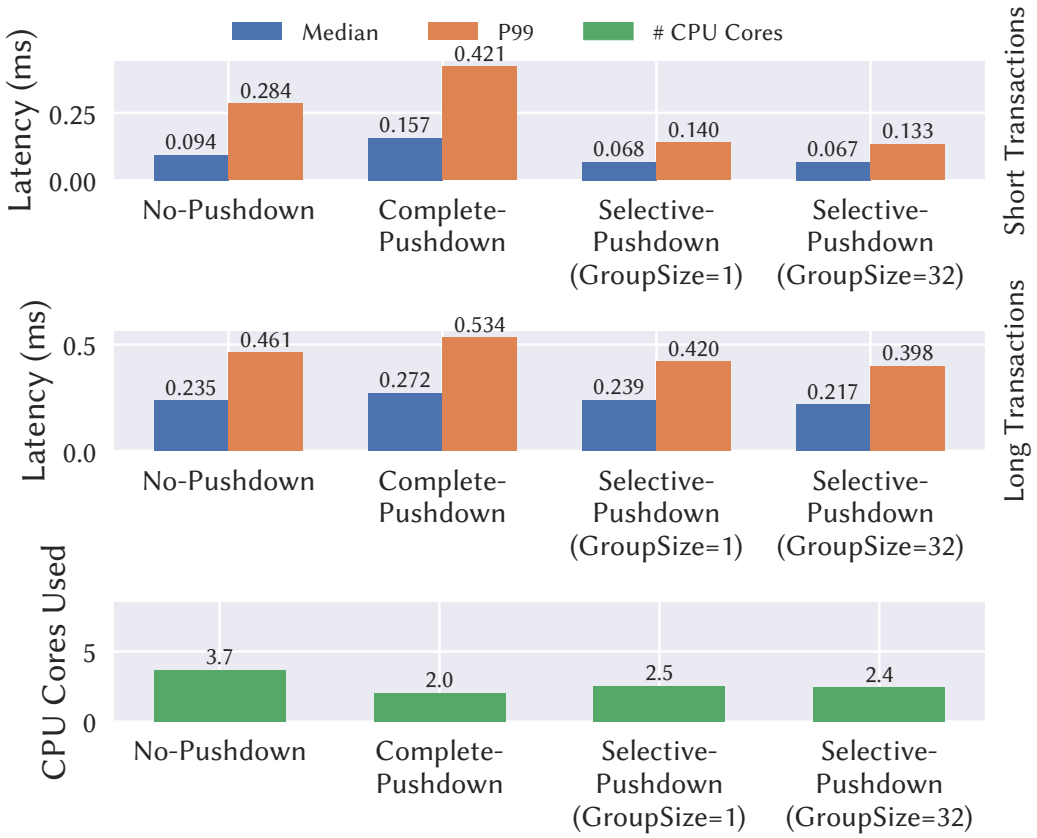



Figure 3.17: **End-to-end Latency of TPC-C on Networked LeanStore** - Selective pushdown with 32-message group results in lowest median/p99 latency for short/long transactions (We use a database of 10 warehouses with an 8GB buffer pool running at ≈ 31,000 transactions/second; We use message-based callback for pushdown mode).

ordered message delivery, and a stream-based interface. To eliminate IPC overhead, we ran Tux in pushdown mode; the callback simply copies data into an application buffer. Stream-based delivery is up to $1.96\times$ slower than the message-based interface because the callback must parse tuples from a byte stream and copy data. Ordered and unordered message delivery have nearly identical costs in our runs: on AWS, packets arrive almost entirely in order for a flow (TCP/UDP), and our implementation detects this case and bypasses the reorder buffer (implemented with std::map in C++). Finally, the Tux-runtime cost increases with tuple size because larger tuples span more packets. Since Tux processes fixed-size packet batches per routine invocation, larger tuples trigger more invocations and context switches. We leave further runtime optimizations to future work.

To quantify the cost of in-order delivery, we simulate packet loss by dropping packets randomly on the sender side which results in receiver side buffering and reordering packets for in-order delivery. The results (Figure 3.16) show significant latency advantages of unordered messaging over ordered delivery under packet loss conditions. At 1% packet loss rate, unordered messaging achieves 42.5%/39.2%/34.1% lower median/P90/P99 latency. At moderate 0.1% loss rates, unordered messaging reduces median/P99 latency by 11.1%/23.7%. This performance improvement stems from unordered messaging's ability to bypass head-of-line blocking and sorting that plague ordered

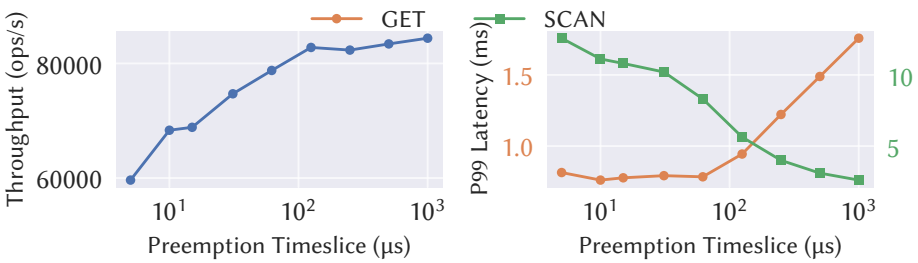



Figure 3.18: **Impact of Preemption Timeslice on Throughput and Tail Latency** - A small preemption timeslice generally results in better tail latency for short operations at the cost of lower throughput and worse latency for long operations (We use a workload of 99% GET and 1% Scan operations with scan length of 5,000.)

protocols during packet loss events.

### 3.5.6 Impact of Preemption Frequency

We next study the impact of preemption on throughput and tail latency. We use a workload that consists of 99% GET and 1% scan operations on the Networked LeanStore server. The scan operation scans 5,000 records. We configure LeanStore to run request processing in message callback executing in the pushdown mode. The results are shown in Figure 3.18. We vary the preemption timeslice from 5$\mu$s to 1ms. Generally, small preemption timeslice results in lower throughput due to signal and context switch overhead. With a 5$\mu$s preemption timeslice, throughput is 40% lower compared to that of 500$\mu$s. After a 100 $\mu$s timeslice, the throughput improvement plateaus as the preemption does not add significant overhead. On the other hand, with a 65$\mu$s timeslice, the p99 latency for GET operation is 2.5$\times$ lower compared to a 1ms timeslice. Smaller preemption timeslice also results in longer tail latency for scan, as short operations are prioritized. However, we find timeslice near 100$\mu$s is a sweet spot that balances throughput, tail latency for short-running callback function, and timely network processing.

## 3.6 Related Work

While Section 3.2 focused on the networking mechanisms and prior designs that directly motivate Tux, this section places Tux in the broader landscape of DB-OS co-design, kernel bypass, programmable networking, and low-latency scheduling.

**DB-OS Co-design** A significant body of research focuses on co-designing database systems to work more efficiently with the underlying OS. libdbos [88] and CumulusDB [89] explore running DBMS in privileged kernel space for designing more powerful abstractions. Another line of research focuses on optimizing scheduling within DBMS engine, including cooperative scheduling using coroutines to hide latency for memory and I/O [76,90–93] and task-based parallelism [94], as well as preemptive techniques like userspace interrupts to prioritize short transactions [95]. While these improve internal efficiency, Tux tackles the complementary bottleneck at the DB-OS network stack/scheduler boundary [9].

**More Kernel Bypass Systems.** IX [50], Arrakis [96] and Pegasus [97] redesign the OS dataplane by removing most kernel-mediated I/O or IPC via user-space data paths, runtimes, and specialized APIs. In contrast, Tux focuses on stock Linux kernels and virtualized NIC hardware in the cloud rather than requiring new OS or kernel modules. MICA [98] holistically co-designs an in-memory KV store and its I/O path to maximize single-node throughput. In contrast, Tux is not an application redesign; it is a transport/runtime substrate that lets DBMSes gain similar benefits via message semantics and selective pushdown. Systems like eRPC [99] provide a high-level RPC interface, whereas Tux operates closer to the transport layer, offering both a compatible byte-stream interface and direct programmability via pushdown. Similarly, while Machnet [100] offers high-performance messaging, its sidecar architecture introduces IPC overhead that Tux's pushdown eliminates, and its in-order messaging can suffer from head-of-line blocking that Tux's unordered messaging avoids. Recent systems like Shinjuku [101], LibPreemptible [102], and Caladan [103] introduce microsecond-scale preemptive scheduling to improve tail latency and resource efficiency with the help of a specialized kernel module or user-space interrupt hardware. Tux can integrate or complement its preemptible runtime with these techniques to reduce preemption overhead.

## 3.7 Summary

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

# Chapter 4

# Practical and Performant Array-Based Translation for Buffer Management

## 4.1 Introduction

The previous chapter showed how database-aware networking can remove a major DB–OS bottleneck without giving up compatibility and deployability. This chapter turns to another OS-facing component on the database critical path: buffer management. A DBMS buffer pool lets a database support data sets larger than physical memory while keeping frequently accessed pages cached in DRAM, making it central to both performance and resource control. When the DBMS's internal components (e.g., query executors, index) access data via logical page identifiers, they use the buffer pool to retrieve an in-memory buffer frame for the requested page. Thus, the core operation of a buffer pool is *translation*: mapping a logical page ID to an in-memory frame. Translation sits on the critical path of nearly every page access, so a buffer pool must keep it fast while retaining fine-grained DBMS control over page I/O and eviction.

Although buffer pool design and implementation are old topics in databases [104], modern workloads have made the translation path more important. Recent high-performance buffer-pool designs [22,31,88,105,106] primarily target OLTP-style workloads dominated by B-tree traversals. The rise of retrieval-augmented generation and semantic search is pushing DBMSs to also support vector search [26,107,108]. These workloads stress the buffer pool in different ways [109]: partition-based indexes are scan-heavy [4,110,111], while graph-based indexes perform irregular, high-fan-out traversals [3,5,112]. The visible cost of translation is roughly inversely proportional to the useful computation performed per page, and is especially exposed in vector search where a single query often spends most of its time accessing hundreds to thousands of pages [109] rather than traversing only a few B-tree levels. Hence, a modern buffer pool must provide fast access across scans, B-tree lookups, and graph traversals, while still integrating cleanly with DBMS data structures.

The central design choice is where to keep the translation state. Existing systems either manage it inside the DBMS or delegate it to OS page tables [24]:

**DBMS-managed:** The most common approach is for the DBMS to maintain the buffer pool's page table in user-space. Most production DBMSs implement the buffer-page translation table as a user-space hash table that maps logical page identifiers to buffer frames [25,26,113,114]. This design is

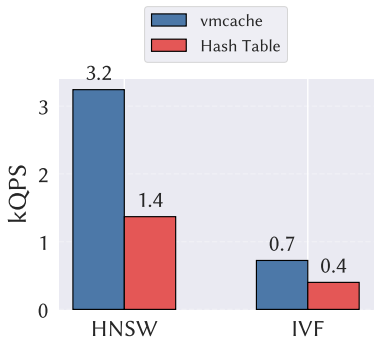



Figure 4.1: **In-memory vector search throughput (SIFT10M)**

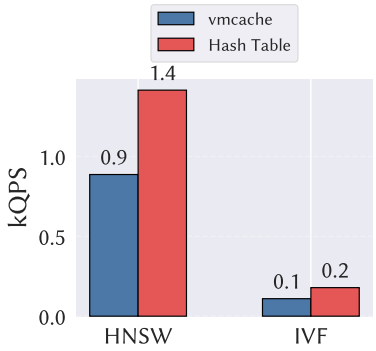



Figure 4.2: **Larger-than-memory vector search throughput (SIFT10M)**

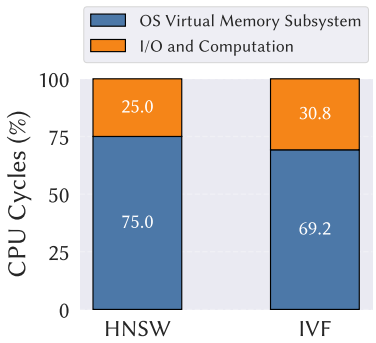



Figure 4.3: **CPU breakdown of vmcache on larger-than-memory vector search (SIFT10M)**

popular because it preserves deployability and keeps eviction and I/O policy under DBMS control, but it introduces significant costs on the translation path: pointer chasing, and synchronization, while hash functions also inherently scatter adjacent page IDs. As a result, scan-heavy workloads lose prefetch efficiency in modern CPU, and high-parallelism graph workloads suffer from reduced memory-level parallelism. Predictive translation can improve hash-table translation [106], but it is workload-dependent and still cannot hide the translation overhead for the scan-heavy and graph workloads. Pointer swizzling can remove software translation [31,105,115], but eviction then has to coordinate with data-structure internals. Before a page can be evicted, the DBMS must find and unswizzle every in-memory pointer to it, which is difficult for structures with cyclic or many-to-many page references such as graphs and doubly-linked B-tree leaves.

**OS-managed:** An alternative is to store the translation state in OS-managed hardware page tables [22,88], where the TLB accelerates resident-page lookup. However, this couples the DBMS with OS page-table management, weakens control over eviction and I/O scheduling, and requires kernel changes for scalable page fault handling [22,88]. A further structural limitation is the *page granularity mismatch*: DBMSs need fine-grained page I/O (4–32KB), yet huge pages (2MB) are essential for TLB efficiency. Because the OS cannot cleanly evict a single 4KB subpage from a mapped 2MB huge page [116–118], these systems must choose between 4KB pages with higher TLB pressure or 2MB pages with larger I/O amplification.

We validate this trade-off empirically on vector search. Figures 4.1 and 4.2 compare vmcache (OS-managed) and a hash-table pool (DBMS-managed) for serving graph-based vector index (HNSW) and partition-based vector search index (IVF) using the SIFT10M dataset in both the in-memory and larger-than-memory scenarios. vmcache outperforms the hash-table pool by 2.3× in memory because the hardware MMU and TLB handle resident-page lookups with no software overhead, but falls behind by up to 1.6× out of memory because vmcache is bottlenecked by the kernel virtual memory subsystem, as indicated in the CPU cycle breakdown in Figure 4.3.

Our broader experimental analysis across scans, B-tree lookups, and graph traversal further shows that no single existing mechanism is preferable for these access patterns while also supporting huge pages and fine-grained DBMS I/O control. These limitations motivate the goal of this chapter: retain the full policy control of DBMS-managed translation state while approaching

Table 4.1: **Key Properties and Workload Efficiency of Buffer-Pool Designs** - Comparison of the three translation-data-structure classes. Workload-efficiency cells report SS, RS, PL, and GT using color-coded badges, where green, yellow, and red indicate strong, mixed, and weak efficiency. GT† in the pointer-swizzling column denotes limited support for graph-style workloads due to multi-parent references. The property columns summarize huge-page friendliness, required OS modifications, I/O control, and memory-overhead scaling.

| Translation Structure | Where Stored | Key Properties | | | | Translation Performance | | |
|---|---|---|---|---|---|---|---|---|
| | | 4KB I/O with Huge-page-backed Frames | OS mods | I/O control | Memory overhead | Baseline | + Predictive Translation | + Pointer Swizzling |
| **Hash Table Translation** | DBMS | yes | none | full | $O$(# cached pages) | SS RS PL GT | SS RS PL GT | SS RS PL GT† |
| **Hardware Radix Page Table** | OS Kernel | no | mixed | mixed | $O$(# storage pages) | SS RS PL GT | \ | \ |
| **Array Translation (Calico)** | DBMS | yes | none | full | $O$(# storage pages) | SS RS PL GT | SS RS PL GT | SS RS PL GT† |

the translation efficiency of OS-managed page-table mechanisms, without invasive changes to data structures or OS kernel modifications.

In this chapter, we present **Calico**, a DBMS-managed buffer pool based on *array translation*: each logical page ID serves as an offset into a translation array that maps to buffer frames. This design avoids hash probing and pointer chasing on the translation path, keeps integration non-invasive via the existing page-based interface, and decouples logical translation from OS page tables. As a result, Calico can back frame memory with 2MB huge pages for TLB efficiency while preserving fine-grained DBMS-managed eviction and I/O. While array translation for buffer management is not new [104], it has seen very limited research and deployment because large, sparse, hierarchical page identifiers [119–123] in modern DBMSs made flat arrays too expensive in terms of memory consumption. Calico makes the approach practical and performant through three techniques: **multi-level translation** with path caching for managing sparse hierarchical IDs efficiently (Section 4.4.1), **active hole punching** to reclaim cold translation memory (Section 4.4.2), and **group prefetch** to hide indirection latency during high-fan-out graph traversal (Section 4.5). On modern hardware, these mechanisms let array translation match OS-based page table translation performance while substantially outperforming hash-table translation across scans, B-tree lookups, and graph traversal.

We implemented Calico as a drop-in replacement for PostgreSQL's buffer manager with about 2.6K lines of code changes. The rest of this chapter first analyzes why existing translation mechanisms fall short across scans, B-tree lookups, and graph traversal, and then shows how Calico decouples logical translation from OS page tables so that the DBMS can combine low-overhead array translation, huge-page-backed frames, and fine-grained eviction and I/O control in a single design. The evaluation shows that Calico reduces translation overhead compared to state-of-the-art designs while remaining competitive with hardware-assisted translation: it improves `pgvector` by 2.9–3.95×, speeds up scan-heavy PostgreSQL queries by up to 3×, and outperforms in-memory-only libraries such as Faiss [124] when data is resident.

## 4.2 Background and Motivation

We now characterize the workload requirements that matter in modern workloads and use them to motivate a design-space decomposition of buffer-pool translation.

### 4.2.1 Workload and Operational Requirements

Translation sits on the critical path of every page access, but different workloads stress it differently. To evaluate designs systematically, we distill modern buffer pool accesses into four patterns that each exercise a distinct aspect of the translation mechanism.

**Sequential scan (SS)** streams through long contiguous page ID ranges (e.g., heap scans, cluster scans in IVF index [4,110]). This is especially important for scan workloads with small per-page computation, such as accessing pre-computed zonemaps or vector distance computation, where translation overhead is not fully amortized by tuple processing.

**Point lookup (PL)** captures latency-sensitive random access such as a B-tree root-to-leaf descent. Each translation is followed by a data-dependent load on the returned frame, so per-access translation latency directly determines throughput.

**Range scan (RS)** begins with a short dependent lookup phase (e.g., B-tree descent) and then scans consecutive leaf or posting-list pages. It combines the dependency-chain sensitivity of PL with the locality demands of SS.

**Graph traversal (GT)** captures irregular, high-fan-out graph traversals such as HNSW beam search [3,5,112]. Expanding a single node may issue dozens of neighbor translations simultaneously, so GT rewards designs that expose memory-level parallelism and penalizes those with serializing dependency chains.

These four patterns span the locality and parallelism axes that modern buffer pools must handle simultaneously. We label them SS, RS, PL, and GT throughout the paper and use them in Table 4.1 and Section 4.3 to expose the trade-offs of each design.

### 4.2.2 Design Space of Buffer Pools

Buffer pool designs can be roughly classified along two orthogonal axes: the **translation data structure** and **where the translation state is stored**. Table 4.1 maps this design space by using these base translation mechanisms as rows. The columns evaluate each design across key system properties and workload behaviors, capturing both their baseline performance and their compatibility with supplementary optimizations (*Predictive Translation*, *Pointer Swizzling*).

**DBMS-managed hash-table pools:** Hash table translation is the most widely used base design in production systems [25,26,114,125,126] where translation state is managed by the DBMS. A hash table maps page identifiers (PIDs) to buffer frames, giving the DBMS control over eviction policies and I/O scheduling. The trade-off is critical-path overhead: collision resolution, synchronization, and pointer chasing create dependency chains that limit memory-level parallelism for high-fanout workloads. Hashing also scatters consecutive PIDs, destroying spatial locality during translation on scans; under high thread counts, lock/atomic synchronization increases cache coherence traffic.

**OS-managed page-table translation:** This approach stores translation state in radix-tree-based page table managed by OS and MMU hardware [22,88,105,127]. *File-backed `mmap`* delegates page

faults, caching, and eviction to the OS [114,128,129], reducing DBMS control over replacement and I/O scheduling. *Virtual-memory-based* systems such as vmcache keep translation state in OS page tables but move policy into the DBMS. In **vmcache** (without exmap), the DBMS performs explicit I/O and eviction using standard kernel interfaces, which preserves user-space deployability but pays higher per-page VM-management overhead such as TLB-shootdown, especially on high-end storage devices. Kernel changes are required to improve scalability of page fault handling and eviction [22,88], but such modifications face deployment barriers.

**Page granularity mismatch:** OS-managed page-table translation fundamentally shares a limitation. DBMSs need fine-grained page management and I/O (typically 4–32KB), while huge pages (2MB) are important for TLB efficiency. When translation state resides in hardware page tables, the OS cannot cleanly evict a single 4KB subpage from a mapped 2MB huge page; it must evict the full 2MB region [116], split the huge page [130,131], or fail/refuse the operation [117,118]. Hence these systems must choose between 4KB pages (fine-grained eviction and I/O, higher TLB pressure) and 2MB pages (lower TLB pressure, high I/O amplification). For this reason, vmcache chooses 4KB pages in favor of fine-grained I/O [22].

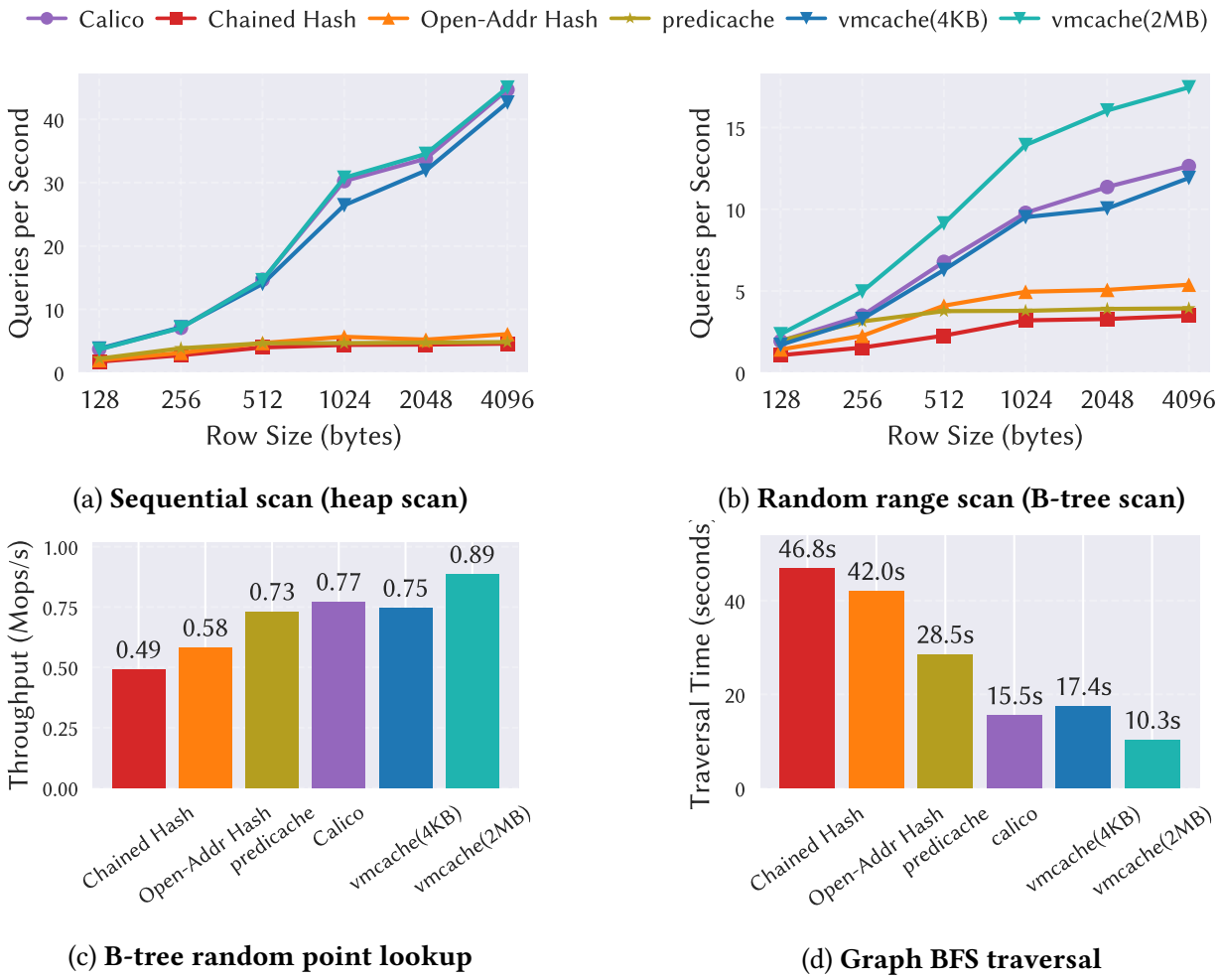
**Pointer swizzling and predictive translation:** Conceptually, they can be applied to any mechanism that exposes a modifiable translation path, but in practice they are most effective for DBMS-managed translation structures; for OS-managed page-table translation, the structure is fixed in hardware/OS page tables. Pointer swizzling [31,87,105,115,132] reduces translation overhead by replacing logical IDs with pointers, but introduces invasive coupling. Evictions require unswizzling/validation bookkeeping, which is hard for multi-parent references (e.g., graph fan-in, sibling links, secondary-index backlinks). LIPAH [133] improves coverage but still requires in-page hints and validation logic. Predictive translation [106] keeps a hash-table translation layer but deterministically assigns storage pages to preferred frame positions so the CPU can speculatively access the likely frame while translation is in progress. It is effective when hot pages remain in preferred positions, but benefits are dependent on workload and buffer-pool size.

**Array Translation:** Array translation uses the page ID directly as an index into a contiguous array of frame references, replacing hash computation and probing with a single indexed memory load. Despite being discussed historically [104], it has seen limited research and deployment because large, sparse page-identifier spaces make a naive flat array impractical. Calico targets this underexplored DBMS-managed design point. The next section performs a microarchitectural analysis to establish whether array translation can match hardware-assisted alternatives.

## 4.3 Translation Performance Analysis

Before addressing sparsity and metadata management in Section 4.4, we first ask whether array-based translation is fast enough to serve as the foundation of a modern buffer pool. To isolate translation cost, we compare four approaches from Section 4.2: hash tables, predicache [106], vmcache [22] (4KB and 2MB pages), and Calico's array-based translation. All buffer pools implement the same interface: given an 8-byte page ID, return a pointer to the buffer frame.

**Setup:** We run on a m7a.8xlarge AWS EC2 instance. All experiments are single-threaded with data fully resident in memory, isolating translation overhead from eviction and synchronization. Page IDs are allocated from 0 sequentially. We report two hash-table baselines: *Chained Hash*




(a) **Sequential scan (heap scan)**

(b) **Random range scan (B-tree scan)**

(c) **B-tree random point lookup**

(d) **Graph BFS traversal**

Figure 4.4: **Impact of Translation Across Workloads** - (a) Sequential scan: Hash tables destroy spatial locality, causing 6.9× slowdown vs Calico Array/vmcache; predicache provides no benefit for sequential access. (b) Random range scan: Gap persists at 2.0× despite random access; predicache underperforms plain hash tables. (c) B-tree lookup: Hash tables still incur clear translation overhead; predicache narrows the gap through speculation, while Calico Array remains faster with a simpler translation path. (d) Graph BFS: Hash-table shows high translation overhead; predicache helps partially. Calico Array matches or outperforms virtual-memory-based translation.

(`std::unordered_map`) and *Open-Addr Hash* (`absl::flat_hash_map`). Chained Hash, Open-Addr Hash, predicache, and Calico Array use OS huge pages to back buffer frames.

We evaluate three workload families that stress different aspects of translation performance: **(1) Scan workloads** (Section 4.3.1) measure how well each mechanism preserves spatial locality for sequential and random page access patterns in analytical queries and partition-based vector search. **(2) B-tree lookup** (Section 4.3.2) evaluates translation overhead for OLTP workloads with dependent memory accesses that limit parallelism. **(3) Graph traversal** (Section 4.3.3) stresses memory-level parallelism with parallel random accesses representative of proximity-graph-based vector search.

Table 4.2: **Microarchitecture Metrics** - Performance counters per page scanned (8KB page, 1024-byte rows) for sum query on 50GB buffer pool scanning 5GB data. Sequential scan: consecutive page IDs (e.g., heap scan); Random scan: non-consecutive page IDs (e.g., B-tree leaf scan). Metrics averaged across all pages scanned.

| Config | QPS | IPC | Inst | LLC Ref | LLC Miss | DTLB Miss | Cycles |
|---|---|---|---|---|---|---|---|
| **Sequential Scan (heap scan)** | | | | | | | |
| Chained Hash | 4.38 | 0.15 | 115 | 21.2 | 3.09 | 1.11 | 755 |
| Open-Addr Hash | 5.67 | 0.21 | 122 | 18.9 | 2.53 | 0.81 | 585 |
| predicache | 4.67 | 0.17 | 118 | 23.5 | 7.55 | 0.54 | 702 |
| Calico Array | 30.3 | 0.77 | 83.8 | 13.8 | 1.33 | 0.002 | 109 |
| vmcache (4KB) | 26.4 | 0.64 | 80.1 | 14.2 | 1.75 | 1.25 | 125 |
| vmcache (2MB) | 30.8 | 0.74 | 80.1 | 14.0 | 1.42 | 0.002 | 108 |
| **Random Scan (B-tree leaf scan)** | | | | | | | |
| Chained Hash | 3.20 | 0.15 | 155 | 34.0 | 10.9 | 2.35 | 1033 |
| Open-Addr Hash | 4.95 | 0.24 | 161 | 29.1 | 8.09 | 1.58 | 669 |
| predicache | 3.78 | 0.14 | 119 | 27.4 | 9.43 | 1.01 | 870 |
| Calico Array | 9.78 | 0.36 | 123 | 22.5 | 5.22 | 0.56 | 338 |
| vmcache (4KB) | 9.52 | 0.34 | 120 | 20.8 | 7.36 | 2.28 | 347 |
| vmcache (2MB) | 13.9 | 0.50 | 120 | 20.7 | 5.01 | 0.56 | 237 |

### 4.3.1 Scans

We measure spatial locality preservation using a scan query on heap pages with row-oriented format (Figures 4.4a and 4.4b, Table 4.2). Sequential scans are common in analytical queries and IVF-based vector search, while random scans occur during index-organized table access. We execute an aggregation query that computes the sum of an integer column over 5GB of data. We vary the row size to simulate different amounts of useful computation per page: smaller rows pack more tuples into each page, increasing the amount of aggregation work amortized over one page translation. *Sequential* scans access consecutive page IDs (simulating heap scans), while *random* scans access non-consecutive page IDs (simulating B-tree leaf scans). This workload stresses whether translation preserves adjacency and lets the CPU prefetch scan streams. Figures 4.4a and 4.4b show Calico's Array reaches 30.3 QPS on sequential scan, 6.9× faster than Chained Hash and 6.5× faster than predicache, while remaining close to vmcache (2MB) at 30.8 QPS. On random scan, Calico Array reaches 9.78 QPS, still 2.0× faster than Open-Addr Hash and 2.6× faster than predicache, and slightly ahead of vmcache (4KB) at 9.52 QPS. Hash-table-based translation scatters translation entries randomly, while array translation keeps entries contiguous, reducing LLC misses and preserving scan throughput. predicache helps less here because speculation cannot recover locality once the translation entries are scattered.

### 4.3.2 Point Lookup

B-tree point queries represent the core OLTP workload. Each lookup traverses from root to leaf, accessing random pages with dependent loads that limit memory-level parallelism. We use YCSB-C (read-only) on 100M records (128 bytes each, 16GB buffer pool, 4KB pages) to evaluate translation overhead in this common access pattern with results in Figure 4.4c and Table 4.3. Chained Hash

Table 4.3: **Microarchitecture metrics per B-tree lookup for YCSB-C workload on 24GB buffer pool with 100M records (128 bytes each)** - Single-threaded read-only point queries with a B-tree depth of 4. Metrics averaged across all lookups.

| Config | Mops/s | IPC | Inst | LLC Ref | LLC Miss | DTLB Miss | Cycles |
|---|---|---|---|---|---|---|---|
| Chained Hash | 0.49 | 0.30 | 1294 | 44.5 | 22.7 | 3.69 | 4,251 |
| Open-Addr Hash | 0.58 | 0.37 | 1327 | 39.3 | 17.8 | 2.18 | 3,583 |
| predicache | 0.73 | 0.45 | 1285 | 28.9 | 16.0 | 1.34 | 2,863 |
| Calico Array | 0.77 | 0.42 | 1143 | 20.7 | 12.2 | 0.46 | 2,707 |
| vmcache (4KB) | 0.75 | 0.40 | 1124 | 24.3 | 14.3 | 2.48 | 2,797 |
| vmcache (2MB) | 0.89 | 0.48 | 1124 | 20.1 | 12.2 | 0.47 | 2,347 |

Table 4.4: **Microarchitecture metrics per graph node visited (including neighbor probes) for BFS traversal on 5M-node proximity graph** - Single-threaded traversal from same starting node. Metrics averaged across all graph nodes visited during full graph traversal.

| Config | Time (s) | IPC | Inst | LLC Ref | LLC Miss | DTLB Miss | Cycles |
|---|---|---|---|---|---|---|---|
| Chained Hash | 46.8 | 0.18 | 3552 | 548 | 337 | 110 | 20,206 |
| Open-Addr Hash | 42.0 | 0.22 | 3937 | 502 | 255 | 91 | 18,107 |
| predicache | 28.5 | 0.36 | 4385 | 307 | 183 | 51 | 12,335 |
| Calico Array | 15.5 | 0.63 | 2248 | 265 | 142 | 60 | 6,671 |
| vmcache (4KB) | 17.4 | 0.28 | 2082 | 261 | 147 | 39 | 7,476 |
| vmcache (2MB) | 10.3 | 0.48 | 2079 | 175 | 114 | 27 | 4,367 |

reaches 0.49 Mops/s and Open-Addr Hash reaches 0.58 Mops/s. predicache improves hash tables to 0.73 Mops/s by overlapping part of hash-based translation with frame access through CPU speculative execution. Calico Array reaches 0.77 Mops/s with fewer instructions per lookup and 24% fewer LLC misses than predicache due to its single-instruction translation and compact array, matching vmcache with 4KB pages.

### 4.3.3 Graph Traversal

Graph BFS traversal exposes translation's impact on memory-level parallelism, which is an important property for modern vector search workloads. We perform breadth-first traversal on a 5M-node graph (simulating HNSW vector index beam search) where each node connects to 44 random neighbors (4KB page per node). When visiting a node, the algorithm probes all neighbors. The workload is deliberately compute-light, so performance is dominated by buffer-pool translation and reveals whether the translation mechanism preserves memory-level parallelism across neighbor probes. The results are shown in Figure 4.4d and Table 4.4. Open-Addr Hash takes 42.0s and predicache reduces this to 28.5s, but Calico Array reaches 15.5s, a 2.7×/1.8× improvement over Open-Addr Hash/predicache, while also outperforming vmcache with 4KB pages. Array translation has two advantages here. First, each translation is a single memory load into a compact, cache-efficient array, so more translations fit per cache line and require significantly fewer instructions compared to hash tables. Second, because each translation is one simple independent memory load, the CPU's out-of-order execution can issue neighbor translations in parallel. This allows Calico to get close to the performance of vmcache with 2MB

pages.

### 4.3.4 Takeaway

Across these workloads, array-based translation matches or outperforms vmcache with 4KB pages. Its advantage over hash tables comes from removing pointer chasing and preserving cache locality during translation. These effects are largest for high-MLP graph traversal and sequential scans, and still visible for point lookups. Array-based translation is therefore a promising design point, but a naive flat array is impractical for sparse hierarchical PID spaces. The next section shows how Calico makes array translation practical and performant.

## 4.4 Calico Design

Although our evaluation in Section 4.3 demonstrated that array translation outperforms hash-table and matches OS-page-table alternatives across sequential, random, and graph-traversal access patterns, a naive flat array is impractical for production DBMSs with large and sparse page-identifier spaces that exceed what a contiguous array can cover efficiently. This section presents the design of Calico, which makes array translation viable.

**Design goals:** Translating the raw performance advantage of arrays into a deployable buffer-pool design requires satisfying four goals simultaneously: ❶ **Sparse PID support** – handle large and sparse page-identifier spaces without allocating a monolithic flat array spanning the entire logical domain; ❷ **Array-fast-path preservation** – retain single-load array translation on the hot path so that the performance gains established in Section 4.3 carry over to the full system; ❸ **Fine-grained DBMS control with huge-page efficiency** – keep eviction, replacement, and I/O scheduling under DBMS control at page granularity while backing frame memory with huge pages for TLB efficiency; ❹ **Active memory reclamation** – reclaim physical memory from cold translation regions so that the translation memory overhead tracks only the working set.

**Overview.** Calico separates DBMS-managed logical translation from OS-managed physical frame backing (Figure 4.5). To handle sparse PID spaces (❶), Calico organizes translation hierarchically, allocating per-region last-level arrays only for active regions; because modern DBMS workloads exhibit strong locality within relations and indexes, a lightweight path cache ensures that the common case reduces to a single array translation (❷). Frame memory is huge-page-backed and managed independently of translation arrays, so page-granularity eviction does not require OS page-table modifications (❸). An active hole-punching mechanism tracks and reclaims translation memory from fully cold regions (❹). The remainder of this section details hierarchical translation and path caching (Section 4.4.1), on-demand array memory management (Section 4.4.2), and the buffer-pool algorithms (Section 4.4.3).

### 4.4.1 Hierarchical Translation and Path Caching

To tackle the problem of large sparse PID spaces, Calico introduces hierarchical translation so memory is proportional to active regions rather than the full logical PID domain. Upper levels map PID prefixes to lower-level translation arrays, and those lower-level arrays are allocated only

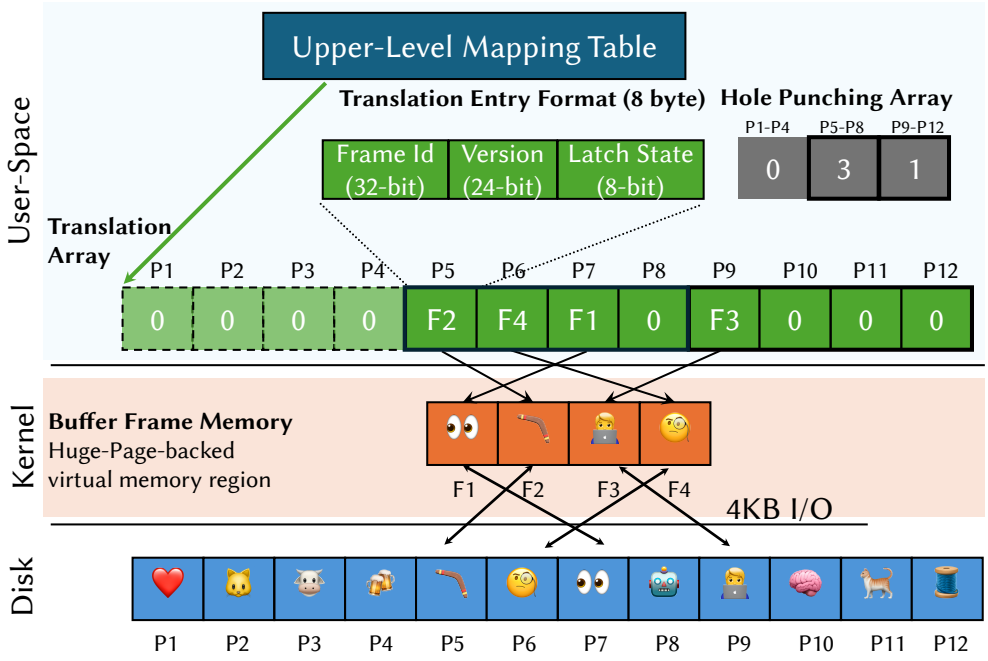



Figure 4.5: **Calico Buffer Manager Architecture** - Pages P5–P7 and P9 are resident in frames F1–F4; all other entries are zero (evicted). The upper-level mapping table resolves prefixes to per-region translation arrays whose 64-bit entries encode frame ID, version, and latch state. Frame memory is huge-page-backed; the DBMS performs 4KB I/O independently. The Hole Punching Array tracks one count per entry group (P1–P4: 0, reclaimed; P5–P8: 3; P9–P12: 1). Groups are shown as 4 entries for readability; in practice each spans one 4KB OS page (512 entries).

when a prefix becomes active. Prefixes with no resident pages keep no materialized last-level array, so empty PID regions do not consume translation memory.

While such multi-level translation introduces extra indirection, the key observation is that modern DBMSs organize page identifiers hierarchically, which induces strong locality in prefix components. B-tree traversals, HNSW traversals, and scans repeatedly access pages within the same relation/index region [107,134–138]. Calico exploits this locality using translation-path caching. Figure 4.6 walks through translation of page identifier $p = (prefix, suffix)$: first check the path cache, then resolve the prefix through the upper-level index on a miss, then perform suffix-based array translation in the resolved last-level array, and finally update the cache with the resolved prefix-to-array mapping. When subsequent accesses reuse the same prefix, upper-level traversal is bypassed and only the final array lookup remains on the hot path. In practice, because successive accesses within a B-tree traversal, HNSW search, or scan almost always share the same prefix, a single thread-local cache entry storing the last resolved (prefix, last-level array) pair suffices to achieve good hit rates.

The remaining design question is how to choose the prefix/suffix split. A natural decomposition follows the storage hierarchy already present in most DBMSs: the *prefix* identifies a stable container region (e.g., database/table/index or relation/fork), and the *suffix* identifies the page within that region. In most DBMS layouts, the suffix is the page or block number within a table/index, which keeps the last-level translation array densely indexable by array offset. Take PostgreSQL

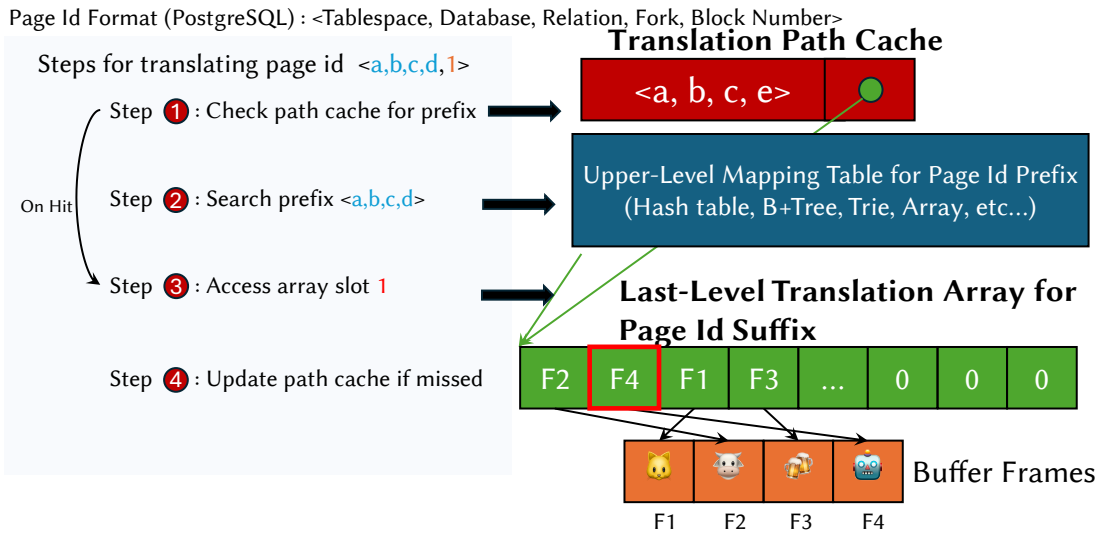



Figure 4.6: **Hierarchical Translation with Path Caching** - Calico decomposes each page identifier into a *prefix* and a *suffix*. The figure walks through four steps: (1) check the translation path cache; (2) on a miss, resolve the prefix through an upper-level index (e.g., radix tree, hash table, B$^+$-tree, or trie) to obtain a last-level translation array; (3) use the suffix to directly index that array on the hot path; and (4) update the path cache with the resolved prefix-to-array mapping.

as a concrete instantiation. For PID `<Tablespace, Database, Relation, Fork, Block>`, Calico uses `prefix = <Tablespace, Database, Relation, Fork>` to select the last-level translation array and `suffix = <Block Number>` for direct indexing in that array (Figure 4.6). This matches observed locality where accesses repeatedly remain in the same relation/fork region while block numbers vary. The same principle generalizes to MySQL's hierarchical `<space_id, page_no>` identifiers. Additional levels can be added when PID-space sparsity requires them.

### 4.4.2 On-demand Array Memory Management

After upper-level prefix resolution, every access goes through a last-level translation array. This hot path must satisfy three requirements: (1) fast array-indexed translation, (2) concurrency control for reads/updates/eviction, and (3) memory usage proportional to the working set despite sparse logical PID spaces.

Calico encodes each `TranslationEntry` as one 64-bit atomic word. The **latch state** (8 bits) supports exclusive/shared synchronization, the **version number** (24 bits) enables optimistic validation for lock-free reads on the buffer frame, and the **frame ID** (32 bits) identifies the buffer frame. Hence, Calico supports addressing 16 TB memory for 4KB page, which is sufficient for most modern workloads. A single load returns both translation and concurrency metadata. Translation is as follows:

```
1 TranslationEntry* te = &TranslationArray[pageId];
2 Frame* frame = frameMem + te->getFrameId();
```

**Memory efficiency and reclamation:** The flat translation array introduces a potential space overhead: for a 16TB SSD with 4KB pages, the array requires 32GB of memory (4G entries ×

8 bytes). CALICO leverages the OS's on-demand memory allocation mechanism combined with **hole-punching** to reduce the *physical* memory usage so that it is proportional to the working set size.

**Zero-Page Copy-on-Write at Startup:** The translation array is allocated via `mmap`. Windows provides analogous virtual-memory reservation APIs [139]. The reserved region is initially backed by a shared zero-filled OS physical page [140]. When a thread first reads an entry, the OS triggers a page fault and establishes a read-only mapping to the shared zero-filled physical page, setting the copy-on-write (COW) bit in the page table entry. Only when an entry is updated (e.g., in `calico_page_fault_handler`) does the OS allocate a physical page, copy the zero page content, and update the mapping. This on-demand allocation pattern aligns with CALICO's access pattern: entries for cold pages that are never loaded remain on the shared zero page indefinitely, consuming no physical memory. Thus, the virtual size of the translation array can be proportional to the logical page ID space, while its physical memory footprint tracks only the working set.

**Zero-Value Entry Design for Lazy Loading:** To exploit this mechanism, CALICO's `TranslationEntry` encoding is carefully designed such that an all-zero 64-bit value represents an *evicted* page state. Specifically, when all 64 bits are zero: (1) the frame ID field is 0, interpreted as `INVALID_FRAME`, indicating no buffer frame is allocated; (2) the latch state field is 0, representing `Evicted`; and (3) the version number is 0. This invariant ensures that at system startup, when the entire translation array is logically filled with zeros (via the shared zero page), every entry correctly represents an evicted page requiring a page fault on first access. Crucially, when a page is evicted (`calico_evict_victim`), CALICO writes a zero value back to the entry, returning it to the invalid state. This allows further memory reclamation by releasing the physical page via hole-punching, as described next.

**Hole-Punching Array:** While lazy allocation via OS demand paging ensures translation arrays consume physical memory proportional to active entries, cold regions in the translation array may remain physically allocated long after pages are evicted. To address this, CALICO introduces the Hole-Punching Array (`HPArray`), a lightweight reference counting structure that tracks the number of valid (non-evicted) translation entries within each OS page of the translation array, enabling active memory reclamation.

The `HPArray` is a flat array of atomic 32-bit counters, where each counter tracks one **entry group**: consecutive translation entries fitting within a single OS page. For a translation array with $N$ entries, `HPArray` requires $\lceil N/\text{entries\_per\_OS\_page} \rceil$ counters. With 512M translation entries and 4KB pages, this requires 1M counters (4MB); with 2MB huge pages, only 2048 counters. The `HPArray` itself is allocated lazily via `mmap`, consuming physical memory only on first write. Each counter reserves one bit as a lock to coordinate hole-punching operations (Section 4.4.3).

**Memory Overhead Discussion:** CALICO's translation array stores one 8-byte entry per 4KB storage page, so its worst-case overhead is proportional to on-storage data size. However, this is less than vmcache [22], which requires ≈16 bytes per 4 KB storage page: 8 bytes of x86-64 page-table entries (unavoidable for any `mmap`-based design) plus an 8-byte page-state. For a 1 TB SSD, CALICO's worst-case translation footprint is 2 GB versus vmcache's ≈4 GB. As flash is roughly $50\times$ cheaper per byte than DRAM [22], CALICO's overhead adds ≈10% to the flash price.

In practice, CALICO is competitive in terms of memory overhead. Real-world DBMS workloads exhibit spatial locality [78], so hot pages cluster into contiguous regions and cold regions form large hole-punchable groups. OS page tables retain non-zero swap entries for evicted pages,

preventing kernel reclamation; Calico's eviction explicitly zeros entries, enabling hole punching to reclaim cold translation memory. Compared with hash tables, the constant-factor trade-off favors Calico on the hot path. While a hash table stores translation entries only for cached pages, it stores both key and value per entry plus load-factor headroom. For example, predicache [106] uses 40-byte entries at a 50% load factor. Calico's 8-byte translation entries are $5–10\times$ denser, directly reducing cache misses during translation (Section 4.3).

### 4.4.3 Buffer-Pool Algorithms

Algorithm 1 presents Calico's core buffer-pool algorithms for pinning, unpinning, page fault handling, and eviction. We assume a thread-local path cache that stores the most recent `(prefix, lastLevelArray)` mapping for each thread.

**Exclusive Pin (Algorithm 1):** To modify a page, a thread must acquire an exclusive lock on the translation entry. To do this, it atomically loads the entry, checks if the page is resident (valid frame ID), and uses compare-and-swap (CAS) to transition from `Unlocked` to `Locked`. If the page is not resident (`INVALID_FRAME`), it triggers the page fault handler. On success, the thread receives a pointer to the frame holding the page data. Note that shared pins can be implemented similarly by storing the number of readers in the latch state of `TranslationEntry`.

**Optimistic Read (Algorithm 1):** For read-heavy operations, Calico exposes a lock-free optimistic read interface. The reader snapshots a `TranslationEntry`, executes the read function on the target frame, and then validates that the entry version and frame ID are unchanged and that the entry is not locked. This avoids atomic pin/unpin traffic on the read path while preserving correctness under concurrent eviction and modifications.

**Exclusive Unpin (Algorithm 1):** Releasing an exclusive pin is a single atomic operation: set the lock state to `Unlocked` and increment the version number. The version bump invalidates any concurrent optimistic readers that observed the old version.

**Page Faulting (Algorithm 2):** When a page is not resident, the fault handler acquires an exclusive lock on the translation entry and double-checks the frame ID (another thread may have already loaded the page). If still invalid, it allocates a frame (from the free list or via eviction), issues I/O to load the page data, then atomically increments the reference count in the `MetadataArray` for the OS page group containing this translation entry. Finally, it updates the entry with the new frame ID and releases the lock. The translation entry lock prevents duplicate I/O when multiple threads fault on the same page, while incrementing the metadata counter before publishing the frame ID ensures the group cannot be hole-punched during page fault.

**Eviction (Algorithm 3):** Calico uses a standard replacement policy (CLOCK) to select a victim page. The algorithm acquires an exclusive lock on the victim's translation entry, writes back the frame if dirty, and invalidates the entry by setting the frame ID to `INVALID_FRAME`. Next, it atomically locks the metadata counter and decrements the reference count for the corresponding OS page group. Crucially, the translation entry is unlocked only *after* acquiring the metadata lock. If the count reaches zero (all entries in the group evicted) after decrement, the thread performs hole-punching via passing `MADV_DONTNEED` hint to OS while still holding the metadata lock, then releases the lock. This ordering ensures that concurrent page faults must wait for the metadata lock before incrementing the counter, preventing any thread from installing a new frame ID in a page being hole-punched.

**Algorithm 1: CALICO Buffer Pool Algorithms**

```
1  Function GET_TRANSLATION_ENTRY(pageId):
2      (prefix, suffix) ← split_pid(pageId)
3      if TLSCache.prefix = prefix then
4          last_level_array ← TLSCache.lastLevelArray
5      else
6          last_level_array ← lookup_leaf_table(prefix)
7          TLSCache ← (prefix, last_level_array)
8      return &last_level_array[suffix]

9  Function PIN_EXCLUSIVE(pageId):
10     while true do
11         te ← Get_Translation_Entry(pageId)
12         old_e ← *te // Atomic load
13         if old_e.frameId = INVALID_FRAME then
14             page_fault_handler(pageId, te)
15             continue
16         if old_e.state = UNLOCKED and
             te.CAS(old_e, (old_e.frameId, old_e.version, LOCKED)) then
17             return frameMem + old_entry.frameId

18 Function UNPIN_EXCLUSIVE(pageId):
19     Get_Translation_Entry(pageId).bump_version_unlock()

20 Function OPTIMISTIC_READ(pageId, read_func):
21     while true do
22         te ← Get_Translation_Entry(pageId)
23         old_entry ← *te // Atomic load
24         if old_entry.frameId = INVALID_FRAME then
25             page_fault_handler(pageId, te)
26             continue
27         if old_entry.state = LOCKED then
28             continue // Spin until unlocked
29         read_func(frameMem + old_entry.frameId)
30         new_entry ← *te
31         if old_entry.version = new_entry.version and
             old_entry.frameId = new_entry.frameId then
32             return // Success
```

**Algorithm 2: CALICO Page Fault Handler**

```
1  Function PAGE_FAULT_HANDLER(pageId, te):
2      while ¬te.try_lock() do
3          continue
4      if te.frameId ≠ INVALID_FRAME then
5          te.unlock()
6          return
7      frameId ← allocate_frame_or_evict()
8      io_read_page(pageId, frameMem + frameId)
       // Atomically increment counter for this entry group
9      groupIdx ← group_index(te)
10     increment_hparray(groupIdx)
11     te.set_frame_and_unlock(frameId)
```

**Algorithm 3: CALICO Eviction with Hole-Punching**

```
1  Function EVICT_VICTIM():
2      victimId ← select_victim_page() // CLOCK, LRU, etc.
3      te ← Get_Translation_Entry(victimId)
4      while ¬te.try_lock() do
5          continue
6      frameId ← te.frameId
7      write_back_if_dirty(frameMem + frameId)
8      te.frameId ← INVALID_FRAME // Zero out frame id
       // Atomically decrement MetadataArray refcount
9      groupIdx ← group_index(te)
10     count ← HPArray(victimId)[groupIdx].LOCK_AND_DEC()
11     te.unlock_evicted() // Now zero out latch state
12     if count = 0 then
           // Last valid entry evicted, reclaim memory
13         madvise(group_base_addr(te), 4096, MADV_DONTNEED)
14     HPArray(victimId)[groupIdx].UNLOCK()
15     return frameId
```

## 4.5 Implementation and Optimizations

This section presents CALICO's group prefetch interface and then describes system integration in PostgreSQL and `pgvector`.

**Algorithm 4: CALICO Group Prefetch Algorithm**

```
1  Function GROUP_PREFETCH(pids, offsets):
2      non_resident_pids ← [ ]
       // Prefetch translation entries
3      foreach pageId ∈ pids do
4          te ← Get_Translation_Entry(pageId)
5          prefetch(te)
       // Prefetch resident pages and collect non-resident ones
6      foreach i ∈ [0, length(pids)) do
7          pageId ← pids[i]
8          offset ← offsets[i]
9          te ← Get_Translation_Entry(pageId)
10         if te.frameId ≠ INVALID_FRAME then
11             prefetch(FrameMemory[te.frameId] + offset)
12         else
13             non_resident_pids.append(pageId)
14     if non_resident_pids ≠ ∅ then
15         io_read_pages(non_resident_pids)
```

### 4.5.1 Group prefetch

Modern workloads like graph-based vector search exhibit predictable access patterns: future page accesses are often known in advance. In proximity-graph-based vector search [3,5,112], when visiting a graph node, the algorithm must probe neighboring nodes whose IDs are stored in the current node. CALICO exploits this predictability through a **group prefetch interface** that issues parallel reads to the translation array and prefetches buffer frame memory. This parallelism operates at two levels:

- **Memory-Level Parallelism**: For resident pages, parallel translation array reads allow the CPU to issue multiple independent loads simultaneously. Modern CPUs can issue parallel memory loads [141–143], hiding memory latency.
- **I/O-Level Parallelism**: For non-resident pages, knowing multiple page IDs in advance enables batched asynchronous I/O, saturating storage bandwidth and reducing total latency.

Algorithm 4 presents the group prefetch algorithm. The interface accepts page IDs with corresponding in-page offsets for targeted prefetching. The algorithm operates in three phases: (1) issues prefetch instructions for translation entries to bring them into cache; (2) issues prefetch

instructions for resident page frames at specified offsets while collecting non-resident page IDs; (3) submits batched reads for non-resident pages. This exposes batching semantics, enabling indexes to exploit modern hardware capabilities.

### 4.5.2 PostgreSQL Integration

We implement multi-level Calico in PostgreSQL v18, which uses a 5-level hierarchical page identifier BufferTag [119]. We use the last-level 32-bit BlockNumber as the suffix and the remaining fields as the prefix identifying the relation. Our implementation uses a hash table for top-level mapping and flat arrays for last-level translation arrays. We extended PostgreSQL's buffer manager with Calico's group prefetch and optimistic read interfaces. The implementation adds approximately 2.6K lines of C code. The modifications are mostly isolated within the buffer manager, preserving compatibility with existing components.

**Translation Path Caching via Thread-Local Storage.** As described in Section 4.4.1, each worker thread caches the most recently resolved (prefix, last-level array) mapping in `thread_local` storage. In PostgreSQL, the prefix corresponds to the upper fields of the `BufferTag`; on each buffer manager call, the thread compares the current tag's prefix bits against the cached value and, on a hit, skips the top-level hash lookup entirely.

**pgvector Integration.** We then modified the `pgvector` extension (about 600 lines) to leverage these interfaces during HNSW graph traversal. PostgreSQL's existing page access requires complex pin/unpin operations using atomic writes that limit memory-level parallelism [144,145]. By using Calico's optimistic read interface, pgvector bypasses these atomic operations. When visiting a graph node, pgvector extracts neighbor page IDs and prefetches neighbors, then Calico uses `calico_optimistic_read` to access neighbor data without pin/unpin overhead. When validation fails, we revert to standard pin/unpin operations to avoid repeated wasted work.

## 4.6 Experimental Evaluation

Our evaluation validates Calico's performance and scalability across modern workloads. We aim to evaluate:

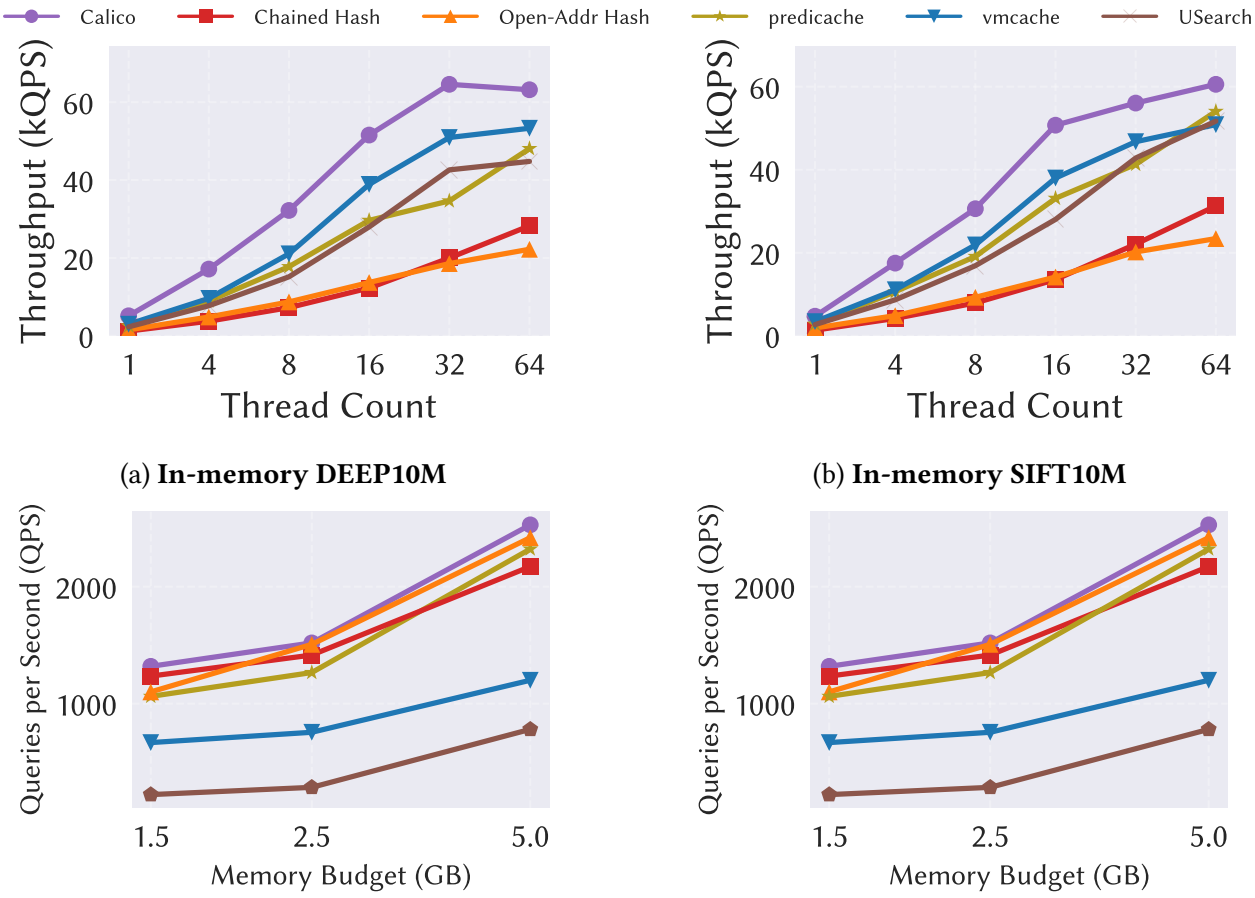
1. Array-based translation overhead compared to mmap, pointer swizzling, hash tables on vector search and OLTP workloads.
2. End-to-end performance gains in PostgreSQL.
3. Memory overhead of array-based translation with hole-punching.
4. Generality of Calico on different CPU platforms.

Unless stated otherwise, all experiments are conducted on a dual-socket server with two AMD EPYC 7513 processors (32 cores per socket, 64 cores/128 threads total, 2.6 GHz base frequency), 504 GB DDR4 memory, and a Samsung PM9A3 3.84TB NVMe SSD (1M random 4KB read IOPS). The system runs CentOS Linux 8.5. We set the CPU governor to performance mode to ensure consistent results. We use Linux transparent huge pages feature via `madvise(MADV_HUGEPAGE)`

hint to enable 2MB huge pages to back frame memory, except vmcache which uses 4KB pages due to the granularity problem. We use huge pages to back hash table memory in hash table-based pools as well.

### 4.6.1 Workloads and Baselines

We evaluate Calico on two workloads:




(a) **In-memory DEEP10M**

(b) **In-memory SIFT10M**

(c) **Larger-than-memory DEEP10M**

(d) **Larger-than-memory SIFT10M**

Figure 4.7: **HNSW Vector Search** - In-Memory and Larger-than-Memory Left: throughput scaling when data fits in memory. Right: throughput under memory pressure across memory budgets at 64 threads.

- **Vector Search**: We evaluate vector similarity search using the HNSW index and IVF index with pruning [111]. We implemented those indexes on buffer manager for accessing pages on storage. We evaluate **Calico**, **vmcache**, **predicache** [106], **USearch** [146] (specialized in-memory HNSW library using `mmap`), and **hash table variants**.
- **OLTP Workloads with B-tree**: We use YCSB-C (read-heavy) and TPC-C (write-heavy) workloads, both operating on B-tree indexes. Baselines include **vmcache** [22] (mmap-based buffer

pool), **predicache** [106], **LMDB** [128] v0.9.31 (a mmap-based key-value store), **WiredTiger** v10.0.2 (hash table), and **LeanStore** [31] at commit 629b41a (pointer swizzling) . We use the lowest isolation level for LeanStore and WiredTiger and disable write-ahead-logging to focus on buffer management performance.

We use baselines that are user-space deployable and do not require kernel modifications, as our goal is to provide a practical solution that can be adopted by existing systems without OS changes.

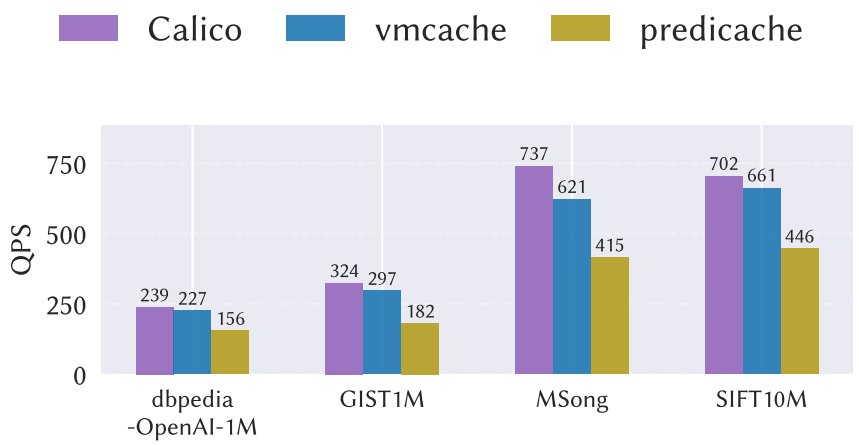
### 4.6.2 Vector Search Workloads

We evaluate CALICO on HNSW-based vector search using DEEP [147] (10M vectors, 96D) and SIFT [148] (10M vectors, 128D). We implement an in-process vector search library with HNSW algorithm [3] and evaluated it on different buffer managers. All experiments run at 64 threads with the same HNSW search parameters (M=16, ef_construction=100, ef_search=50). We enable optimistic reads and group prefetch for all buffer managers. For all hash table-based pools except the lock free hash table, we partition the hash table and use a latch per partition to reduce contention when accessing the hash tables. We enable the group prefetch technique for all buffer managers except USearch to utilize I/O parallelism of SSD. USearch relies on kernel paging which does not provide an interface for group prefetching.

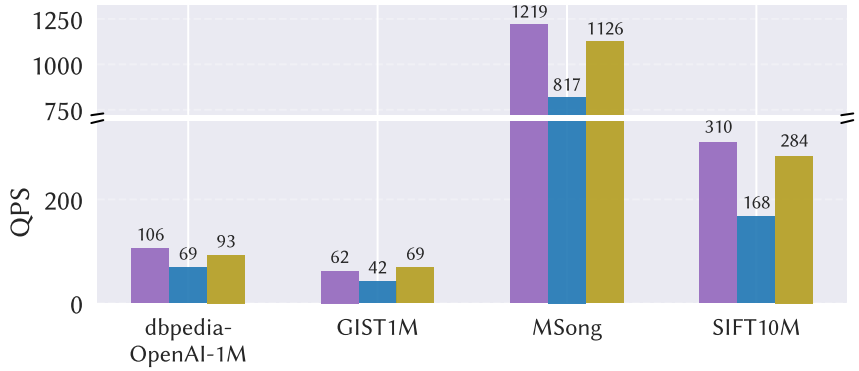
**In-Memory HNSW.** Figures 4.7a and 4.7b show throughput scaling when the working set fits entirely in memory with a buffer pool size of 12GB. At one thread, CALICO has the largest single-core advantage: it is 1.72×/1.38× faster than vmcache and 2.23×/1.74× faster than USearch on DEEP10M [147]/SIFT10M [148]. Hash table variants (except predicache) are significantly slower than virtual-memory-based approaches and CALICO due to translation overhead and synchronization costs. While predicache outperforms plain hash tables both in terms of performance and scalability via lock-free hash tables and speculative execution, it is still outperformed by CALICO by up to 2× due to mispredictions. At 64 threads, the gap between CALICO and other systems narrows as they approach the memory bandwidth limit of the system.

**Larger-than-Memory HNSW.** The results in Figures 4.7c and 4.7d show that when working sets exceed available memory, CALICO outperforms both vmcache and USearch by up to 2–6×. The performance gap stems from fundamental differences in I/O management. USearch relies on `mmap` with synchronous page faults: when data is not resident, the OS blocks threads until I/O completes, serializing execution. At 1.5GB memory, USearch only achieves 189-222 QPS (6× worse than CALICO). All other buffer managers (CALICO, vmcache, and hash tables) use prefetch interfaces to issue parallel I/O requests, avoiding blocking page faults. This explains USearch's steep degradation under memory pressure. Among prefetch-enabled systems, CALICO outperforms vmcache by up to 2×. The root cause for this difference is TLB shootdown overhead. When vmcache evicts pages via `madvise`, the OS interrupts all 64 threads to invalidate TLB entries, which is a synchronization bottleneck that serializes execution. CALICO avoids this problem because it evicts pages in user space without OS involvement, eliminating TLB shootdowns for data pages.

**IVF Vector Search.** We also evaluate a partition-based IVF vector search index [111] on top of buffer managers on four datasets: dbpedia-OpenAI-1M (1536D), GIST1M (960D), MSong (420D), and SIFT10M (128D). IVF vector search probes a set of cluster partitions via sequential scans, stressing spatial locality rather than graph-style random access. We partition each dataset into




(a) **In-memory**



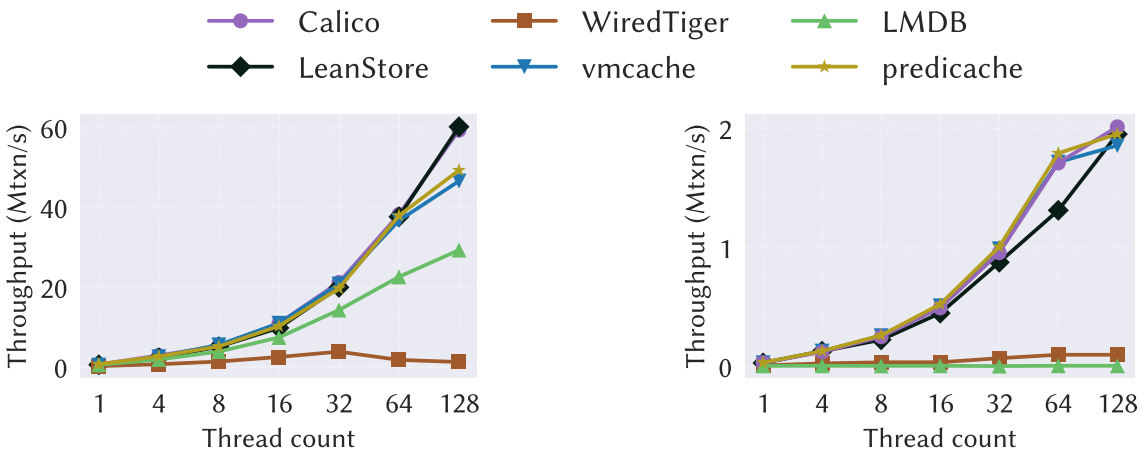

(b) **Larger-than-memory**

Figure 4.8: **IVF Vector Search Performance** - QPS for IVF scan-based vector search on four datasets.

$\sqrt{N}$ clusters where N is the number of vectors in the dataset and each query probes 16 of them. We use one thread for the in-memory scenario to isolate translation overhead without synchronization noise, and 64 threads for the larger-than-memory scenario to saturate SSD bandwidth and expose system-level bottlenecks such as eviction and TLB shootdown costs. We configure the buffer pool size so that it is able to cache 30% of the dataset for the larger-than-memory scenario. Figure 4.8a shows that in in-memory scenario, Calico outperforms vmcache by 1.05–1.19× and predicache by 1.53–1.78× across the four datasets, consistent with the sequential-scan advantage established in Section 4.3.1. Under memory pressure at 64 threads (Figure 4.8b), Calico now outperforms vmcache by 1.48–1.85× across all four datasets, because vmcache's per-page eviction triggers frequent cross-core TLB shootdowns that dominate at high thread counts. Against predicache, Calico is similar in throughput as they use the same user-space I/O subsystem.

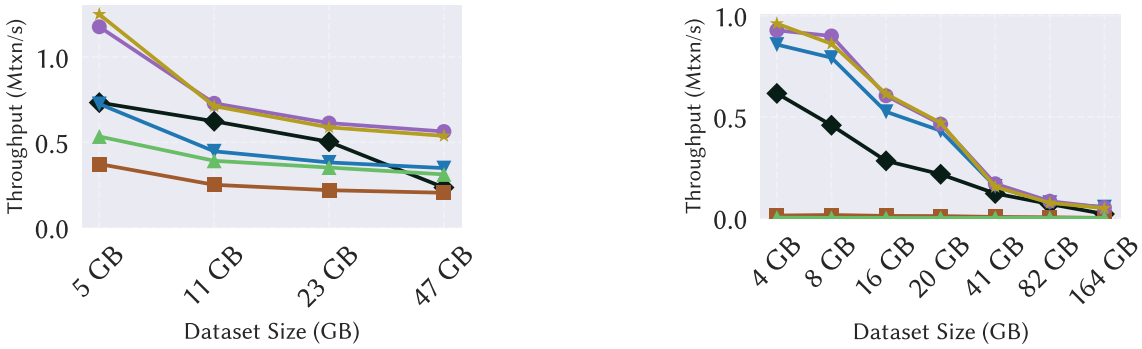
### 4.6.3 OLTP Workloads

We evaluate Calico on YCSB-C and TPC-C using the B-tree index in two scenarios: **in-memory** and **larger-than-memory** OLTP.




(a) **YCSB-C** (b) **TPC-C**

Figure 4.9: **In-memory YCSB-C/TPC-C Throughput Scaling** - YCSB-C: 100 M entries=12.8 GB, 16 GB buffer pool; TPC-C: 200 warehouses=40 GB, 80 GB buffer pool

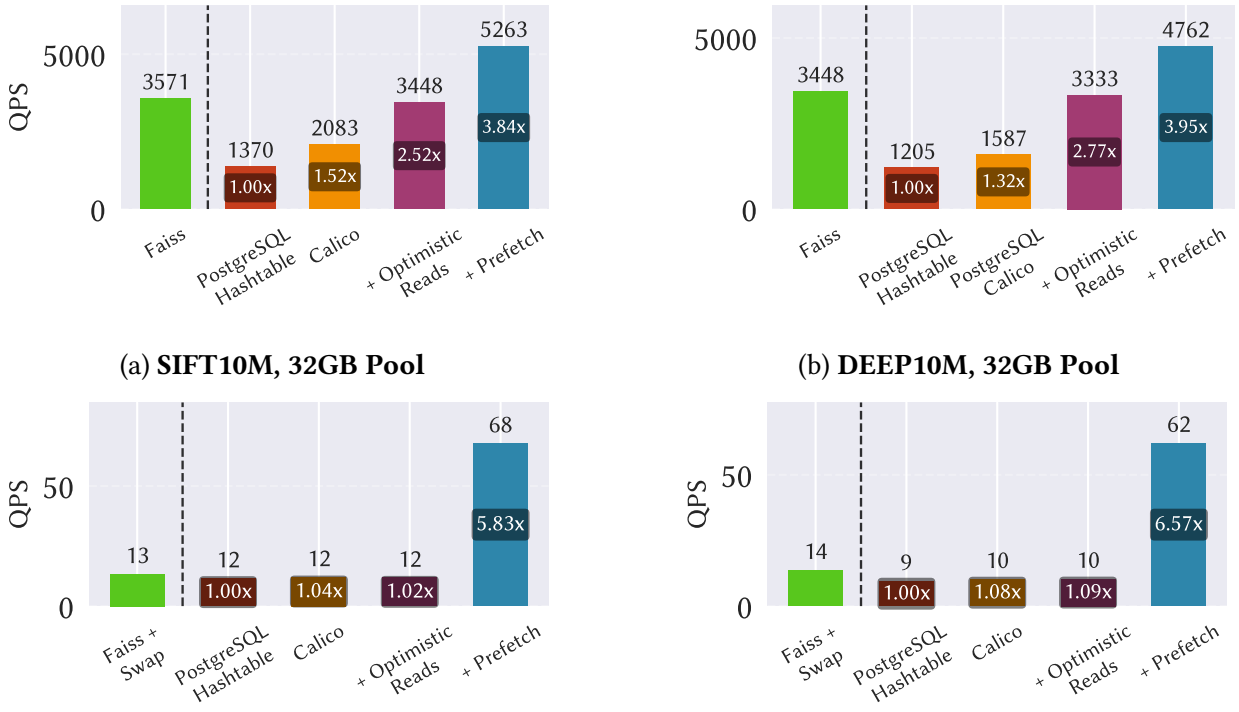



(a) **YCSB-C** - Throughput (b) **TPC-C** - Throughput

Figure 4.10: **OLTP Workloads (Larger-than-Memory)** - Measured throughput for YCSB-C (4 GB buffer pool) and TPC-C (8 GB buffer pool).

**In-Memory.** Figure 4.9 shows throughput scaling from 1 to 128 threads. YCSB-C uses a 16GB buffer pool, while TPC-C uses an 80GB buffer pool. For YCSB-C (Figure 4.9a), Calico starts slightly behind predicache at low thread counts but scales better, reaching 1.27× vmcache and 1.20× predicache at 128 threads. For TPC-C (Figure 4.9b), Calico, predicache, vmcache, and LeanStore remain in the same performance tier, with Calico slightly leading at 128 threads. The main takeaway is that Calico matches state-of-the-art OLTP throughput even on write-heavy workloads, without relying on pointer swizzling or prediction. LMDB and WiredTiger remain much slower because their designs do not scale well under heavy write contention. This demonstrates that Calico's array translation is very competitive for OLTP compared to state-of-the-art systems.

**Larger-Than-Memory.** Figure 4.10 compares systems as dataset size exceeds buffer pool capacity (4GB for YCSB-C, 8GB for TPC-C) with 64 threads. For YCSB-C (Figure 4.10a), Calico is competitive at the smallest dataset size and then becomes the best system as the dataset size grows. It




(a) **SIFT10M, 32GB Pool**

(b) **DEEP10M, 32GB Pool**

(c) **SIFT10M, 2GB Pool**

(d) **DEEP10M, 2GB Pool**

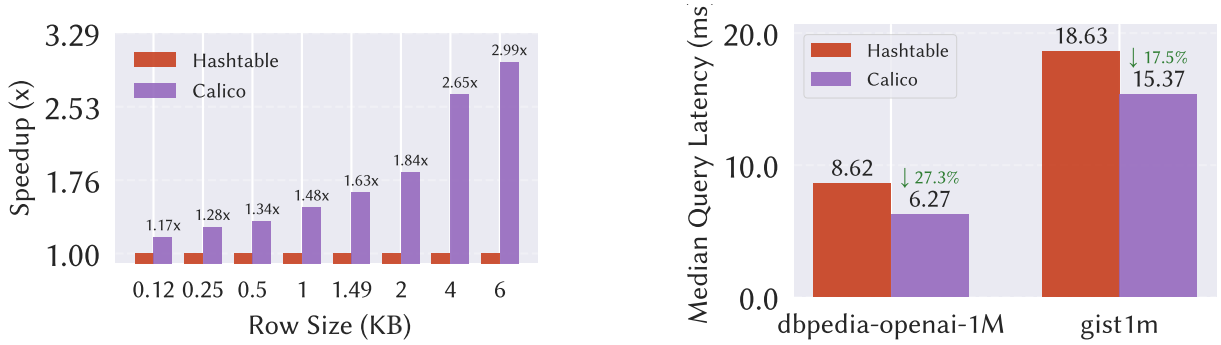
Figure 4.11: **PostgreSQL `pgvector` HNSW Performance** - Bars show query throughput as CALICO optimizations are incrementally enabled on top of the PostgreSQL hashtable baseline (ef_search=64). Top row: data fits in memory (32GB pool); bottom row: index exceeds memory (2GB pool). Faiss (in-memory library) is included as an upper-bound reference.

consistently outperforms vmcache by about 1.6× and remains ahead of predicache at larger dataset sizes. The key reason is that vmcache is bottlenecked by the kernel under memory pressure as each evicted page triggers an expensive TLB-shootdown. CALICO avoids that bottleneck by keeping translation and I/O control in user space. WiredTiger remains lower throughput because its 32KB pages amplify I/O cost once the dataset exceeds buffer pool size. For TPC-C (Figure 4.10b), CALICO, vmcache, and predicache remain close in performance, with CALICO usually at or near the top. This shows that CALICO preserves state-of-the-art OLTP performance not only in memory but also out of memory. LeanStore degrades more noticeably as the dataset grows, while LMDB remains fundamentally constrained by its single-writer design.

### 4.6.4 PostgreSQL Integration

We next evaluate CALICO's impact on PostgreSQL v18 using two workloads: (1) vector similarity search via the HNSW index with `pgvector`, and (2) heap scans with varying row sizes to simulate different amount of computation per page. We use a single PostgreSQL client in Python for the experiments.

**Vector Search.** Figure 4.11 shows query throughput for SIFT10M and DEEP10M datasets using `pgvector` with HNSW index. We start with vanilla PostgreSQL buffer manager and incrementally




(a) **Scan-Heavy Query Speedup**

(b) **IVFFlat Query Latency**

Figure 4.12: **Impact of Sequential Scan on PostgreSQL** - Left: relative speedup of Calico over PostgreSQL Hashtable for `SUM` sequential scan, where Hashtable is normalized to 1.00×. Right: IVFFlat query median latency on DBPedia OpenAI-1M (1536D) and GIST1M (960D) datasets.

enable array indexing, optimistic reads, and group prefetch. We additionally compare with Faiss [124] (in-memory-only vector search library) with the same parameters. The cache hit rate of the translation path caching is close to 100% because PostgreSQL stores all pages of the HNSW index within the same relation.

For SIFT10M (Figure 4.11a), Calico array translation achieves 1.52× speedup over the PostgreSQL hashtable baseline. Adding optimistic reads improves to 2.5× speedup by eliminating atomic pin/unpin overhead and cache coherence traffic [144,145]. Full Calico with group prefetch reaches 3.84× speedup, demonstrating that workload-aware prefetching exploits memory-level parallelism. For DEEP10M (Figure 4.11b), the improvements are similar: array translation achieves 1.32× speedup, optimistic reads reach 2.77× speedup, and full Calico with group prefetch reaches 3.95× speedup.

**Larger-than-memory.** The bottom row of Figure 4.11 shows performance with a 2GB buffer pool, where the HNSW index significantly exceeds memory. In this I/O-bound scenario, base Calico and optimistic reads provide modest improvements (1.04–1.09×) since I/O latency dominates. However, group prefetch achieves dramatic speedups: 5.83× for SIFT10M and 6.57× for DEEP10M. Calico's prefetch API effectively hides I/O latency by exploiting HNSW's graph structure to issue parallel I/O requests. We also run Faiss in a Linux cgroup of 2GB memory to enforce the same memory limit, allowing OS paging to occur. Since Faiss relies on virtual memory for paging which is synchronous, it cannot exploit the batch group prefetch in Calico for better I/O parallelism. Therefore, Faiss was outperformed by up to 5.2×.

**In-Memory Sequential Scan.** Figure 4.12a reports sequential `SELECT SUM(column)` scans on 15GB of data with row sizes from 0.125 to 6 Kb (128 to 6144 bytes). The figure is normalized to PostgreSQL Hashtable (1.00× baseline). Calico outperforms Hashtable at every row size, with speedup increasing from 1.17× (128B) to 2.99× (6144B), and 1.70× geometric-mean speedup across all row sizes. This trend matches the locality analysis in Section 4.3: hash-table translation scatters consecutive page IDs across buckets, while Calico keeps adjacent translation entries contiguous and preserves prefetch-friendly access. Figure 4.12b extends the comparison to IVFFlat, a partitioned vector index that linearly scans pages within each cluster. On DBPedia OpenAI-1M [149] (1536D) and GIST1M (960D) datasets with a 32GB buffer pool, Calico reduces median

query latency by 27.3% and 17.5%, respectively, at identical Recall@3 (0.91 and 0.79). The buffer-manager share of execution time drops from 15–28% (Hashtable) to 1–4% (Calico). This shows that translation overhead in vector search and analytics can consume a nontrivial amount of execution time in a production-grade database system, and Calico's array-based design effectively eliminates that bottleneck to improve query latency.

### 4.6.5 Translation Memory Overhead

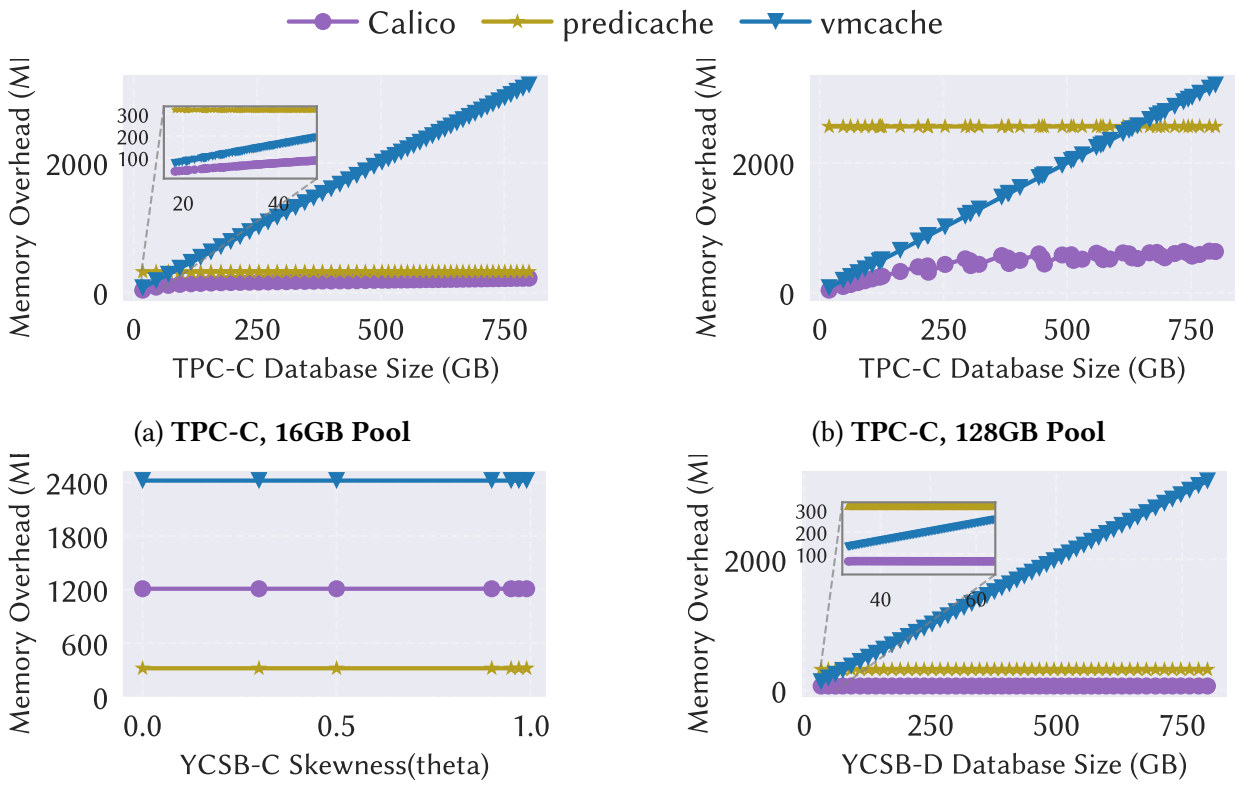
We evaluate translation memory consumption using TPC-C (100 warehouses, 18GB–800GB), YCSB-C (614GB), and YCSB-D (800GB, read-latest workload). We compare **Calico** with hole punching, **predicache**, which uses a lock-free hash table but stores 40-byte translation entries, and **vmcache**. Buffer pool configurations are 16GB and 128GB for TPC-C, and 16GB for YCSB-C and YCSB-D. predicache's translation memory consumption scales with buffer pool size rather than database size. For a buffer pool with $N$ frames, predicache provisions its table at $N \times 40 \times 2$ bytes, where 40 bytes is the entry size and the factor of 2 maintains a 50% load factor. We account for *all* memory used for translation state. For predicache this includes the hash table itself. For vmcache this includes OS page tables in addition to the state array used to track resident pages. For Calico, we include the translation array and the metadata for hole punching.

Figure 4.13 shows translation memory consumption as database size grows. predicache remains constant at 320/2560MB for 16GB/128GB pools. vmcache grows linearly with database size. Calico's hole-punching mechanism reclaims memory for cold regions by exploiting temporal locality. For TPC-C with a 16GB pool, Calico reclaims up to 86.2% of translation memory, which is below predicache's 320 MB despite covering the full logical page space. At the 128GB pool, reclamation drops to 60.4%, but Calico still uses only 632.80 MB, far below predicache's 2,560 MB. YCSB-D shows the strongest hole punching. As new records become the hottest pages, older insertions turn cold and Calico reclaims 95.7% of the translation memory. This is 0.21× predicache and 46.9× smaller than vmcache. In contrast, YCSB-C is the worst case. Its read-only workload and Zipfian distribution spread hot keys throughout the key space, preventing hole punching from forming contiguous cold regions. Nevertheless, Calico is only half of vmcache's memory footprint.

### 4.6.6 Ablation Study

We conduct an ablation study using the DEEP10M dataset and HNSW vector search to quantify the incremental benefits of Calico's optimizations when data fits in memory, shown in Figure 4.14. Array translation delivers the primary gain by 1.59×, increasing throughput to 4.9 kQPS. Backing frame memory with 2MB huge pages adds a further 1.35× speedup by reducing TLB pressure, though applying huge pages to the compact translation array itself provides no additional benefit. Optimistic reads further improve performance to 7.7 kQPS by removing atomic reference counting overhead and enabling memory-level parallelism. Lastly, group prefetching reaches 9.8 kQPS by hiding memory latency during graph traversal and increasing memory-level parallelism.

**Why Group Prefetch Helps Array Translation More.** To isolate the impact of group prefetch, we run an in-memory graph BFS benchmark on a 5M-node graph and compare no prefetch against group prefetch for Calico's array translation, open-address hash translation, and predicache. Table 4.5 shows that group prefetch materially helps only array translation, while hash-based




(a) **TPC-C, 16GB Pool**

(b) **TPC-C, 128GB Pool**

(c) **YCSB-C(614GB), 16GB Pool**

(d) **YCSB-D, 16GB Pool**

Figure 4.13: **Translation Memory Overhead** - Calico's hole punching keeps translation space proportional to working set in TPC-C and YCSB-D. YCSB-C shows challenging behavior with spatially dispersed hot keys that prevent hole punching, yet Calico still uses 0.50× the memory of vmcache.

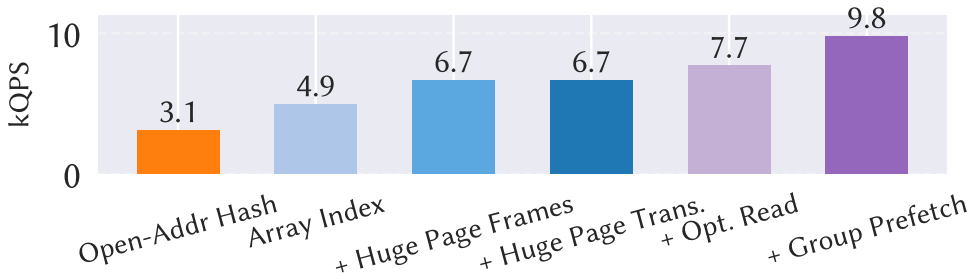
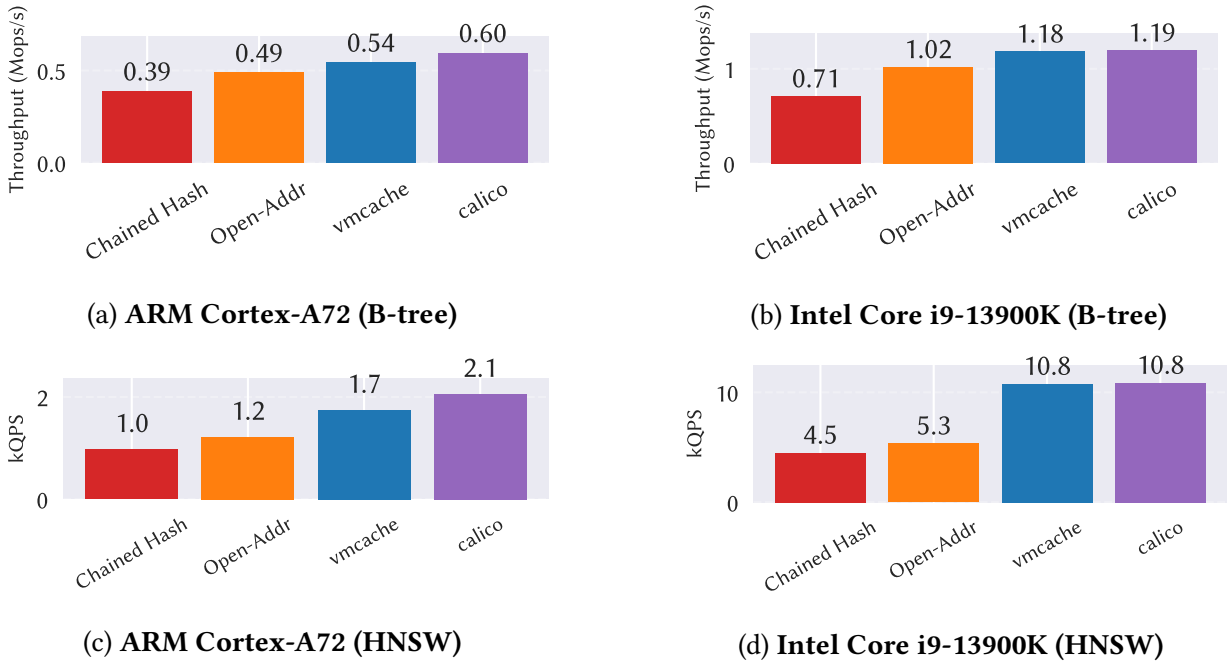



Figure 4.14: **Ablation Study (HNSW DEEP10M, 0.88 Recall@10)** - Cumulative speedup of Calico optimizations. Array indexing provides the largest gain (1.59×), followed by incremental benefits from huge pages, optimistic reads, and prefetching, totaling 3.16× speedup over baseline.

designs and vmcache see little or negative gain. For Calico, prefetch lowers cycles, L3 stall time, and LLC misses enough to produce a clear end-to-end speedup. In Calico, each PID resolves to one array entry, so high-fan-out graph traversal exposes many independent translation loads that software prefetch can overlap. In hash-table and predicache designs, each prefetch still incurs

Table 4.5: **Microarchitecture Analysis of Group Prefetch** - In-memory graph BFS microbench on a 5M-node graph, comparing no prefetch vs. group prefetch for various buffer pool designs. Group prefetch helps only CALICO (1.201×); hash-based and vmcache variants show little or negative gain. Per-node counters are reported as No→Yes prefetch, and speedup is computed from average traversal time.

| Mechanism | Speedup | Cycles/op | L3-stall/op | LLC-miss/op |
|---|---|---|---|---|
| CALICO | 1.201× | 4903→4082 | 5521→4271 | 84.3→65.8 |
| Open-Addr Hash | 1.008× | 11774→11680 | 13736→12835 | 158.7→205.8 |
| predicache | 0.946× | 11201→11840 | 11310→11870 | 309.9→173.9 |
| vmcache(4KB) | 0.916× | 5410→5905 | 6295→6650 | 93.7→98.0 |
| vmcache(2MB) | 0.940× | 3657→3878 | 3958→4037 | 63.0→71.9 |




(a) **ARM Cortex-A72 (B-tree)**

(b) **Intel Core i9-13900K (B-tree)**

(c) **ARM Cortex-A72 (HNSW)**

(d) **Intel Core i9-13900K (HNSW)**

Figure 4.15: **Cross-Platform Validation using Single-threaded B-tree and HNSW vector search** - CALICO consistently outperforms hash tables and matches or exceeds vmcache across ARM and Intel platforms.

hash table probing and possible collision handling before reaching translation state. These extra instructions and control dependencies reduce effective memory-level parallelism, so the overhead of prefetch is not repaid by comparable latency hiding. vmcache likewise benefits little because translation state lookup is not the dominant dependent bottleneck on this workload.

### 4.6.7 Cross-Platform Validation

To validate that CALICO's benefits generalize across CPU architectures, we conducted single-threaded experiments on ARM Cortex-A72 (16 cores, 32GB RAM) and Intel Core i9-13900K (24 cores, 64GB RAM) using HNSW vector search with SIFT1M dataset and YCSB-C workloads with

100M 128-byte records.

**HNSW Vector Search** (Figures 4.15c and 4.15d): On ARM, CALICO achieves 2 kQPS, outperforming vmcache by 18% and chained hash by 2.1×. On Intel, CALICO reaches 10.8 kQPS, matching vmcache and exceeding chained hash by 2.4×. The larger ARM gap (18% vs. 1%) suggests ARM is more sensitive to TLB overhead.

**B-tree YCSB-C** On ARM (Figures 4.15a and 4.15b), CALICO achieves 0.6 Mops/s, outperforming vmcache (4KB pages) by 10% and open-addr hash by 22%. On Intel, CALICO reaches 1.19 Mops/s, exceeding open-addr hash by 17%. CALICO's huge pages (2MB) eliminate TLB pressure that vmcache faces with 4KB pages, while direct array indexing consistently outperforms hash tables (17–22%) across both shallow (ARM) and deep (Intel) pipelines.

## 4.7 Related Work

In this section, we discuss related work in the areas of main-memory database systems, indirection and virtual memory, software prefetching, larger-than-memory vector search, and OS-level memory management.

**Main-Memory DBMS and Buffer Management.** Main-memory database systems [27,28,150,151] manage data at tuple granularity and have been extended to larger-than-memory datasets [127,152,153]. Anti-Caching [152] keeps indexes in memory while evicting cold tuples, limiting scalability [154]. Recent buffer management research explored persistent memory [155,156], remote memory [157–159], translation with SIMD hash table [160], or improving memory utilization under skew [161,162]. CALICO complements these approaches: its direct array translation delivers near-hardware translation performance at page granularity, enabling fine-grained eviction control without kernel modifications and without abandoning the buffer manager.

**Indirection and Virtual Memory.** Prior systems use indirection arrays, mapping tables, or virtual-memory mechanisms at different layers and for different objectives. Bw-tree [163,164] and Bf-tree [165] maintain index-local mapping tables from logical tree node IDs to memory pointers or disk offsets to support latch-free updates, delta chains, or mini-page buffering. Those live mappings remain non-zero, so their arrays scale with index size and cannot exploit CALICO's all-zero evicted state for lazy allocation, shared-zero-page backing, or hole punching. ERMIA [166] uses indirection arrays for record-level MVCC by mapping object IDs to version chains. RUMA [167] rewires OS page mappings to build flexible memory regions for resizing, partitioning, and snapshotting. CALICO instead provides a DBMS-wide user-space buffer-pool translation layer that is agnostic to data structure design, enabling broad applicability across diverse workloads.

**Software Prefetching.** Software prefetching has been widely studied to hide memory and IO latency in various contexts, including database query processing [90,91,168] and transaction processing [76,92]. Unlike existing work, CALICO's group prefetch interface allows modern workloads such as graph-based vector search to express memory-level/IO parallelism to the buffer manager.

**Larger-Than-Memory Vector Search.** Parallel SSD I/O has been exploited to improve disk-based vector search [5,110,169,170]. Unlike existing work, CALICO focuses on improving in-memory performance of buffer-managed vector search index and I/O performance under memory pressure through group prefetch.

**OS-Level Memory Management.** Recent research has explored improving memory management through OS-level mechanisms. Enhanced OS paging systems [171–177] aim to improve performance for larger-than-memory workloads, but operate at hardware page table granularity and face the eviction granularity problem. DBMS/OS co-design approaches such as libdbos [88] and CumulusDB [89] explore running database systems in privileged kernel space to enable more powerful abstractions for buffer management and snapshotting. In contrast, Calico demonstrates that careful user-space design can achieve hardware-page-table performance with fine-grained eviction without kernel modifications.

## 4.8 Summary

We present Calico, a DBMS-managed buffer pool that replaces hash-table and page-table translation with a compact array, combining huge-page-backed frames, multi-level translation, hole punching, and group prefetch into a single design that is strong on sequential scans, random lookups, high-parallelism graph traversals, and larger-than-memory I/O while remaining deployable in user space and non-invasive to existing data structures. Our evaluation shows that Calico delivers 2–6× higher throughput than page-table-based designs under memory pressure, 1.6× speedup over hash tables on B-tree workloads, and up to 3.95× in-memory and 6.5× out-of-memory end-to-end speedup in PostgreSQL/pgvector, achieving hardware-competitive resident-page performance while retaining full DBMS control over eviction and I/O.

# Chapter 5

# Practical DB-OS Co-Design with Privileged Kernel Bypass

## 5.1 Introduction

The previous two chapters showed that user-space co-design can remove major DB–OS bottlenecks without modifying the host kernel: CALICO gives the DBMS control over buffer management while preserving fast translation, and TUX removes kernel networking overhead while retaining kernel-driver compatibility. However, some performance-critical mechanisms remain out of reach from user space. Page tables, TLB management, privilege levels, and interrupt delivery are all controlled by privileged hardware in the kernel and exposed to applications only through interfaces such as POSIX. For database systems, this boundary is increasingly limiting: the DBMS often knows the semantics of its snapshots, buffer-pool mappings, isolation domains, and scheduling decisions more than the OS, but cannot directly act on the hardware mechanisms that implement them.

Virtual-memory snapshotting is a concrete example of why direct control over privileged mechanisms matters. HyPer [27] used `fork` to isolate OLAP queries from OLTP execution by running analytical queries on a forked copy of the OLTP process. Redis similarly relies on `fork` to create background checkpointing and persistence processes while the main server continues serving requests. In both cases, the database wants a fast snapshot of its address space. Linux, however, exposes this capability through a general-purpose process snapshot. The DBMS must therefore pay for page-table copying, copy-on-write bookkeeping, and kernel policies whose semantics are designed for Unix processes rather than database snapshots. HyPer eventually moved away from this approach to software-based MVCC because the OS mechanism was too expensive and too hard to control.

Buffer management exposes the same limitation from another direction. `mmap` and the Linux page cache provide hardware-assisted address translation to user space, but they also give the kernel control over eviction and page-fault handling, which is problematic for high-performance DBMS buffer management [24]. Modern DBMSes therefore often fall back to software-managed page caches, giving up the hardware translation path that would otherwise be attractive.

Existing DB–OS co-design approaches do not provide this combination of control and practicality. Custom database OSes [11,12] and unikernels [89] can expose privileged mechanisms directly to the DBMS, but they require moving the database into a new OS environment and giving

up the mature Linux ecosystem of drivers, tools, and operational practices. Kernel modules, new system calls, and specialized kernel subsystems [21,22,178,179] preserve Linux as the host OS, but they place DBMS-specific code inside the kernel, raising security and maintainability concerns and coupling the system to unstable kernel internals. eBPF-based approaches [46–48,68,180] are easier to deploy, but their restricted programming model limits the design space. User-space bypass avoids host-kernel changes, but it cannot directly manipulate page tables, TLBs, privilege levels, or interrupt delivery, which are the mechanisms needed for the snapshotting and buffer-management use cases above.

This chapter explores a different point in the design space, where one database process gains safe access to privileged hardware mechanisms without abandoining Linux ecosystem or rewriting the DBMS around a new OS. We introduce **co-design with privileged kernel bypass**. The database runs in the kernel space of a lightweight guest environment on top of a hypervisor, giving it direct access to virtualized privileged hardware such as privilege levels, page tables, IOMMU state, and interrupts. At the same time, the host Linux kernel remains intact and continues to provide the normal process abstraction and POSIX ecosystem. We build this model using a process-oriented hypervisor based on Dune [181], which exposes Intel VT-x support to a single Linux process and proxies system calls back to the host kernel.

We implement this paradigm in LibDBOS, a minimal guest kernel for database systems. LibDBOS is not a replacement for Linux; it is a way to specialize the small set of privileged mechanisms that matter to the DBMS while leaving the rest of the host OS ecosystem in place. To demonstrate the benefit, this chapter presents two virtual-memory mechanisms enabled by privileged kernel bypass. Snappy is an instantaneous high-frequency snapshotting mechanism that replaces the general-purpose `fork` path with a DBMS-tailored virtual-memory subsystem. Tabby is an in-kernel buffer pool that uses virtual-memory hardware for page translation while preserving DBMS control over eviction and avoiding costly TLB shootdowns and unscalable memory allocation. Together, these mechanisms show what becomes possible once database semantics can safely reach privileged hardware.

The rest of this chapter develops this privileged kernel-bypass approach. It first describes the LibDBOS execution model and the minimal guest-kernel support needed to run existing DBMS code while exposing database-specific privileged mechanisms. It then presents two mechanisms built on top of this approach: Snappy, an instantaneous high-frequency virtual-memory snapshotting mechanism, and Tabby, a hardware-translation-based buffer pool that preserves DBMS eviction control. The evaluation shows that privileged kernel-bypass co-design can deliver substantial performance gains for snapshotting and buffer management while retaining host-kernel compatibility.

## 5.2 Background and Motivation

This section focuses on the mechanisms that directly motivate LibDBOS, including virtual-memory abstractions, the limits of existing DB-OS co-design approaches, and the virtualization background needed for the rest of the chapter. Broader related work on DB-OS co-design, kernel bypass, virtual memory, snapshotting, and buffer management is discussed in Section 5.8.

### 5.2.1 DB-OS Interface Mismatch

First, we will discuss several applications of virtual memory abstraction for database systems and highlight the mismatch between OS virtual memory interfaces and database and the benefits of DB-OS co-design.

**Virtual Memory Snapshot**. A useful OS abstraction is memory snapshotting or `fork` in Unix. This has been used in several database applications. For example, Redis employs `fork` to create a snapshot of the process containing the in-memory database, which is then serialized to storage as checkpoints. Similarly, the HTAP database system HyPer [27] exploited `fork` to run OLAP queries on a snapshot of an in-memory OLTP database, facilitating workload isolation. All these systems rely on the efficient implementation of `fork`, which is typically achieved through the Copy-on-Write (CoW) technique on modern OSes. However, `fork` is synchronous and single-threaded in Linux, and its latency grows linearly with the memory footprint of the forked process, as shown in Figure 5.1. Notice the latency can be as noticeably high, even for a medium-sized memory footprint (tens of GBs). To get a consistent memory snapshot, the application needs to quiesce latency-sensitive threads for query processing (Redis) and transaction processing (Hyper). Therefore, such high latency introduces significant stalls and tail latency. Figure 5.3b illustrates that the tail latency of Redis write queries drastically increases when there is an ongoing checkpoint using `fork`. Furthermore, such high fork latency forces systems to take snapshots in a frequency of seconds (Hyper) to minutes (Redis). This might be OK for checkpoint purposes (Redis). However, for Hyper-like HTAP databases, such low frequency means some OLAP queries are served on very stale data. Therefore, a low-latency and high-frequency snapshot mechanism would enable a VM-snapshot-based HTAP application with much fresher data.

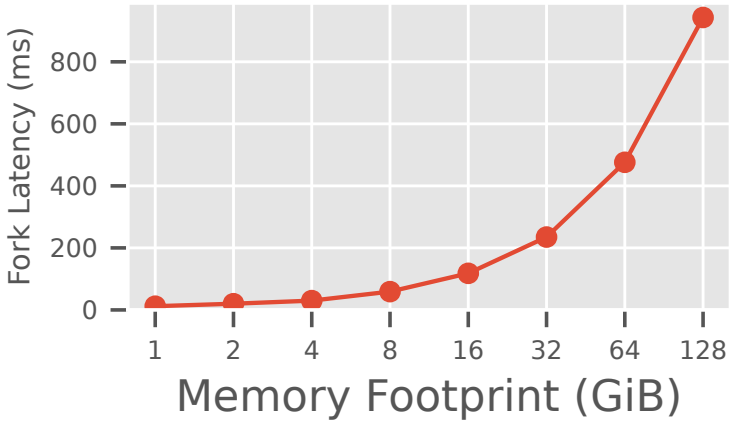



Figure 5.1: **Fork Latency varying Memory Footprint**

**fork Bottleneck Analysis**. Intuitively, most of the time should be spent copying the page table for a CoW-based fork implementation. With a 64 GB memory footprint, the page table is roughly 128 MB, assuming 4 KB page and 8-byte last-level PTE (page table entry). Note that this experiment is run on a machine with a measured 8 GB/s bandwidth for sequential memory access with a single thread. It should only take around 16 milliseconds for such copy in theory. However, it took about 500 milliseconds, shown in Figure 5.1. To find out why, we measured the benchmark program's cycle distribution during the `fork` system call using `perf`. To our surprise, 98% of

```
Percent (cycles)
        ...
17.95               mov    (%rax),%rax
0.31                test   $0x80,%ah
                    jne    4e9
24.03               2f8:   lock   incl   0x1c(%rdx)
0.09                mov    (%rdx),%rax
                    test   $0x80000000,%eax
1.49                je     316
        ...
21.01               316:   lock   incl   0x18(%rdx)
        ...
17.85                370:   lock   andb   $0xfd,(%r12)
```

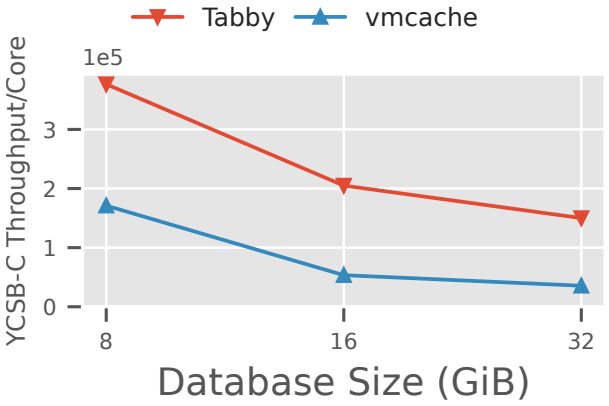
Figure 5.2: **CPU Cycle distribution in `copy_pte_range` in Linux** - most of the cycles are spent in memory accesses to update reference counts

the cycles are spent in `copy_pte_range` kernel function during the sampling period. Figure 5.2 illustrates the hot spots in the assembly code. By analyzing the source code, we found that these hot spots are memory accesses to increment the reference count of the physical pages pointed by PTEs in the page table. While copying the page table is mostly sequential memory access, the metadata of physical pages are grouped sequentially in physical memory address order, resulting in random memory access. These reference counts are critical per-page metadata for maintaining the life cycle of physical pages needed to implement various Linux VM features such as shared memory, page cache, swapping, and memory-mapped files. Instead, with our proposed snapshot mechanism without reference counts, we can drastically improve the latency of creating a snapshot in Redis, shown in Figure 5.3.

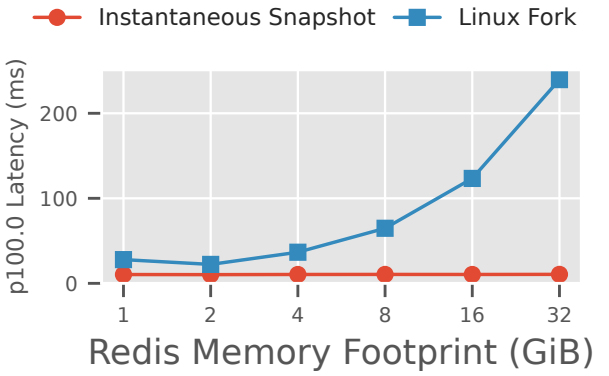
**MMAP-based Buffer Management**. Some database systems utilize `mmap`-based APIs for file system I/O and database buffer management. Such an approach exploits the virtual memory's capability to address memory larger than the physical memory capacity. It relies on the OS (e.g., Linux kernel) for caching database file pages. Compared to a traditional buffer pool that manages a pool of hot pages (identified by logical page id) in memory, a mmap-based approach eliminates one layer of indirection (e.g., software hash table) using virtual memory hardware and improves performance when the working set fits in memory and the workload is read-only.

However, as pointed out recently [24], `mmap` VM APIs exposed by traditional OSes have major problems when data does not fit in memory, or the workload is not read-only. One major bottleneck is TLB shootdown, which occurs when the kernel changes a page-table entry and other CPU cores may still have the old virtual-to-physical translation cached in their local translation lookaside buffers (TLBs). To prevent those cores from using stale translations, the kernel sends inter-processor interrupts and waits for them to flush the affected TLB entries. This synchronization can become a serious scalability bottleneck for buffer management on modern storage devices, because fast storage makes page eviction and remapping frequent enough that shootdown costs become visible [24]. These problems also include page-allocation overhead and the lack of APIs for implementing ARIES-style transactional safety. vmcache [22] showed that we can control page eviction and I/O in user space while still leveraging virtual memory hardware for translation through clever use of `mmap` APIs. However, without modification to the kernel memory subsystem, vmcache is still bottlenecked by the Linux kernel memory allocator and TLB-shootdown when manipulating virtual memory at high speed. Figure 5.3a illustrates this point: on a YCSB-C

out-of-memory workload, the throughput per core of vmcache is substantially lower than our co-designed buffer pool TABBY as vmcache spends 70% of the cycles doing TLB-shootdowns in the Linux kernel.




(a) **Buffer management on fast SSDs (64 Threads, 8GB Buffer Pool)**




(b) **Tail latency of write queries on Redis during checkpoint**

Figure 5.3: **Benefits of DB-OS Co-design**

| | Specializing Linux Kernel | User/Kernel Bypass | Unikernel on VM | Privileged DB Process |
|---|---|---|---|---|
| **Co-design Space** | Medium | Medium | Large | Large |
| **Compatibility** | High | Medium | Medium | High |
| **Ecosystem Connectivity** | High | High | Low | High |
| **Security** | Low | High | Medium | Medium |
| **Maintainability** | Low | High | High | High |

Table 5.1: **Qualitative Comparison of DB-OS Co-design Approaches**

### 5.2.2 Why Are Alternatives Not Sufficient?

Next, we discuss existing solutions and explain why they do not sufficiently address the challenges raised in Section 5.1, and summarize their strengths and weaknesses in Table 5.1.

**Specializing Linux Kernel.** Most co-design follows this approach. For example, the `exmap` module in vmcache [22] specializes Linux memory subsystem to provide efficient allocation and batching TLB-shootdowns. Anker [21] added a system call to get around the limitation of `mmap` for fine-grained memory snapshots. However, fundamental design principles adopted by the Linux kernel limit what could be modified. For example, it is practically impossible to eliminate the page reference count without a major re-design of the Linux virtual memory subsystem. Hence, the co-design space is constrained by the legacy code and assumptions of Linux. Furthermore, the Linux virtual memory subsystem is complex and requires significant familiarity with the Linux kernel to ensure the modified system is safe and functional.

**Kernel Bypass.** Another general approach for removing OS overhead is bypassing the OS kernel, implementing all performance-critical (e.g., file systems and network stacks) components

in the user space, and using I/O libraries such as DPDK/SPDK. However, since these I/O libraries run in user space, they cannot program the interrupt hardware. Therefore, they use spin-polling to complete requests, which is inefficient when there is no load. Similarly, there is no way to directly program the virtual memory hardware in user space. We have to adopt a software-based translation table for the buffer pool or complex pointer-swizzling techniques. This results in sub-optimal performance and increased complexity for the database system. Overall, kernel bypass might leave many performance/simplicity opportunities on the table.

**User Bypass.** Recent work [46] also explored pushing frequently-used data-intensive components (e.g., B-tree traversal and network connection balancing) into the kernel space by leveraging eBPF infrastructure. One major research challenge is that eBPF only supports user programs that are verifiably safe. For example, loops must be bounded, and the programs must use a restricted set of kernel data structures for stateful operations. The Linux kernel also restricts the use of floating-point operations, which are essential for many database applications. This essentially requires rewriting existing database systems using a restricted programming model that only works in the Linux kernel.

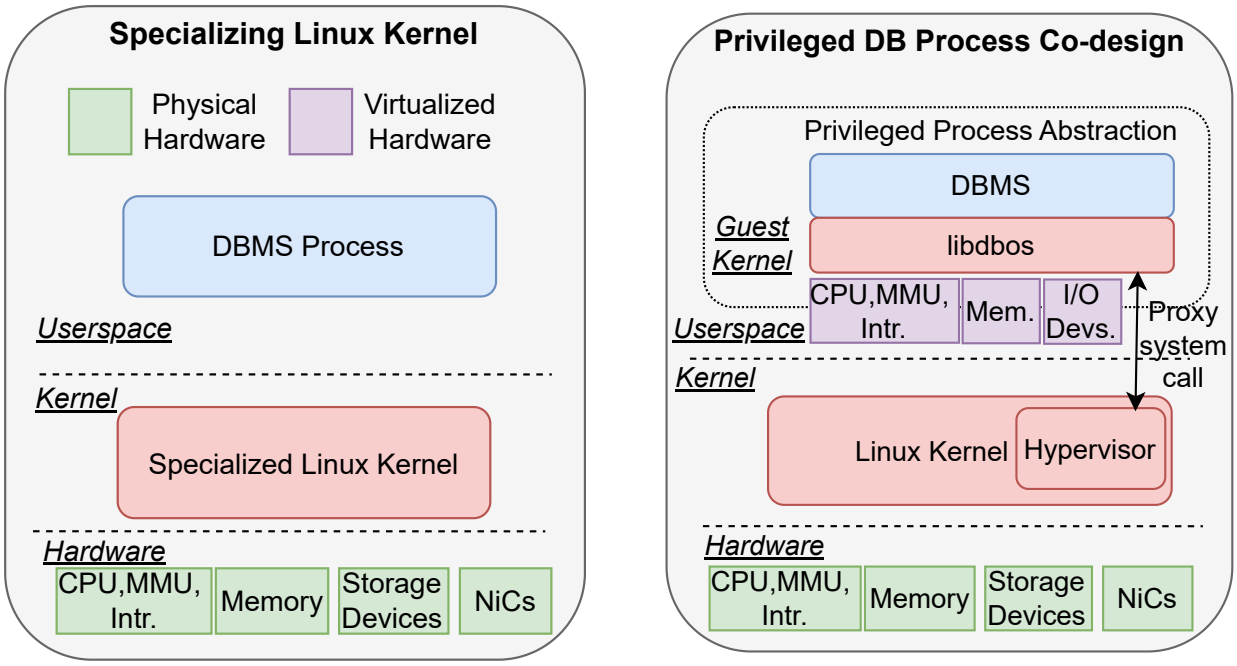
**Co-design with Unikernel.** A recent work [89] envisions running DBMS in a Unikernel on top of a VM in the cloud. A Unikernel [19,182,183] is a type of minimalistic OS kernel that runs only one application in a single address space and privilege level. Such a minimal OS is suitable when running on a VM where the hypervisor handles deployment and isolation. A direct implication of this design is that the application has direct access to all hardware and privileged instructions. Under this deployment, the database runs at the same privilege level as the kernel. Therefore, system calls are ordinary functions served directly by the guest kernel. However, Unikernels require the implementation of drivers for virtualized hardware and POSIX compatibility for databases dependent on POSIX APIs. Furthermore, the tooling and ecosystem around Unikernels remain underdeveloped.

### 5.2.3 Virtualization

**Hardware Assisted Virtualization.** While the mismatch between DBMS and OS interfaces has been observed decades ago and still exists, virtualization has become a cornerstone for modern data centers and cloud vendors during the past decades, revolutionizing the management and deployment of computing resources. By multiplexing a physical machine into multiple virtual machines, virtualization enhances resource utilization, security isolation, and deployment ease. Over the past decades, virtualization has significantly improved in terms of performance. Hardware-assisted virtualization, such as Intel VT-x and AMD-v, enables virtual CPUs to run at native speed. Extended Page Tables (EPT) improve memory management performance by reducing expensive round-trips to the hypervisor for page fault handling. IO-virtualization (e.g., SR-IOV) allows a physical device, like a NIC or SSD, to appear as multiple separate physical devices to be passed-through to the guest, bypassing the expensive I/O stack of the host OS and software-based I/O virtualization. Previous studies [184–189] found that running database systems in virtualized environments only imposes a small overhead for both OLTP and OLAP workloads. Furthermore, most recent cloud database offerings [43,190,191] are deployed in virtualized environments, supporting the case that the virtualization layer does not impose significant performance overhead.

**Privileged Process via Virtualization.** Another novel way of leveraging virtualization is to make a Linux process privileged, pioneered by Dune [181], which is a hypervisor running as a

Linux kernel module and safely exports the Intel VT-x virtualization support to a Linux process. Processes in Dune mode have direct access to virtualized hardware such as MMU, page tables, interrupt, and privileged instructions for manipulating them. Unlike a Unikernel, Dune proxies system calls made by the process to the host kernel. Therefore, full application compatibility and the Linux ecosystem can be preserved. This is the key tool we leverage for our co-design.




(a) **Specializing Linux Kernel**

(b) **Privileged DB Process Co-design**

Figure 5.4: **Comparison of Co-design Architectures**

## 5.3 Privileged Kernel-Bypass Co-Design

In this section, we describe and analyze high-level paradigms for DB-OS co-design. Figure 5.4 contrasts the traditional co-design approach of specializing the Linux kernel with the proposed privileged DB process approach. In traditional co-design (Figure 5.4a), the database runs as a process in user-space while the kernel is modified to support database-specific functionalities. The database process is, therefore, restricted to a set of unprivileged instructions. It interacts with the kernel through system calls to request privileged resources. As Table 5.1 shows, specializing the kernel directly has significant security and maintainability issues, which may explain why this approach has not gained widespread adoption.

Figure 5.4b illustrates privileged DB process co-design. In this method, the database still runs as a Linux process. However, the database process can unidirectionally enter virtualization mode with the help of a type of special hypervisor, Dune [181,192], running as a module in the Linux kernel. Dune exposes a process abstraction for the application rather than a virtual machine abstraction. However, in the environment, the database process obtains direct access to

the virtualized hardware provided by the hypervisor. This is facilitated by LibDBOS, a small kernel built on top of libdune [181] with enhancements and DB-OS co-designed techniques.

LibDBOS inherits from libdune capabilities that allow DBMS developers to customize the logic of various exception handlers (page fault/interrupt/trap), manipulate page table and TLB cache. LibDBOS adds the ability to send inter-processor interrupts and specialize guest kernel memory allocation. At the program's start, developers may override the default handlers provided by LibDBOS. For example, by default, LibDBOS proxies system calls from the DBMS to the host kernel as is. DBMS developers may interpret system calls from the DBMS by registering a custom system-call entry handler. Developers may also override the default page fault handler with DB-OS co-designed page fault handling. This override capability is the key infrastructure for realizing Snappy (Section 5.4) and Tabby (Section 5.5) that will be covered next.

Building upon these customizable capabilities, DBMS developers can also use a more holistic approach to optimize performance through selective co-design between the database and operating system, which we describe next.

### 5.3.1 Workflow of Selective Co-design

In this section, we describe the systematic workflow DBMS developers can follow to implement selective co-design. The first step is identifying bottlenecks in the host kernel, such as the virtual memory subsystem in Linux during snapshotting operations described in Section 5.2. Once these bottlenecks are identified, the next step involves specializing specific functionality in LibDBOS to bypass the slow one in the host kernel while continuing to use other components of the host kernel. For instance, Snappy (Section 5.4) creates a streamlined and simplified virtual memory subsystem tailored for snapshotting in LibDBOS, bypassing the Linux kernel's snapshotting process but still utilizing key virtual memory features like the page cache, shared memory, and memory-mapped files.

**Taking Advantages of Linux Kernel Advances.** Previous research has demonstrated that the Linux storage stack, using modern I/O interfaces such as `io_uring`, can achieve impressive scalability, reaching tens of millions of IOPS on NVMe SSD arrays [193,194]. Leveraging this, DBMS developers can reuse the Linux storage stack for I/O while optimizing performance-critical subsystems. This selective integration is made possible by privileged kernel bypass, which allows for the redirection of system calls to the host kernel. Consequently, we can harness Linux's advanced I/O capabilities without the overhead of developing a new I/O stack and drivers. The same principle applies to networking. Chapter 3 uses Linux's eBPF/XDP interfaces as the kernel advance that enables a high-performance networking fast path, while LibDBOS targets the privileged virtual-memory mechanisms that eBPF/XDP cannot expose. These techniques address different OS bottlenecks and can be combined rather than treated as mutually exclusive alternatives. Our kernel buffer pool, Tabby (Section 5.5), seamlessly reuses the Linux I/O path while bypassing the slow virtual memory manipulation.

Privileged kernel bypass also remains compatible with existing kernel-bypass mechanisms because it preserves the process abstraction. A DBMS running on LibDBOS is still a Linux process that can link against user-space libraries, share memory, use file descriptors, and issue system calls when needed. As a result, a deployment can combine privileged mechanisms for CPU and virtual-memory control with user-space bypass mechanisms such as DPDK and SPDK for accessing storage devices directly.

### 5.3.2 Practical Considerations

**Security and Maintainability**. While the privileged DB process still requires a hypervisor in the host kernel and possibly introduces security vulnerabilities, we argue that the hypervisor is a relatively thin and stable software layer with a much smaller attack surface than various specialized host kernel subsystems for database applications. Therefore, it is much easier to maintain the hypervisor module than a set of ad-hoc changes in the host kernel.

**Deployment Options.** There are several options for deploying a privileged DB process. **On premise**, users have control over physical hardware and can install the DBOS hypervisor for privileged DB process. Another option is to leverage **bare-metal cloud instances**, which are widely available in all major cloud vendors [195–197] where users can install the DBOS hypervisor for maximum performance. Finally, one can leverage **nested virtualization** which is also supported by many cloud instances. The hypervisor can operate in a virtualized environment with some performance overhead.

### 5.3.3 Comparison to Unikernel-based Co-design

Unikernel-based co-design and privileged DB process co-design expose a similar co-design space because both allow the DBMS to execute with kernel-level privilege, although both provide weaker security isolation than unprivileged bypass mechanisms because DBMS code runs in a privileged context. However, Unikernel-based co-design presents a radical all-or-nothing trade-off. Applications must entirely move to a different ecosystem and infrastructure to obtain the benefits. For example, OSv, a relatively mature Unikernel project started 10 years ago, has less than 10 contributors [1] over the last two years. In contrast, Linux had thousands of contributors in the last release cycle. [2]. Tooling such as Unix shell/utilities based on multi-process OS, which is familiar to DBAs and is relied on by many DBMS features (backups, monitoring, troubleshooting), is unavailable in Unikernel due to its single-process design principle. Co-design based on the privileged DB process, in this regard, does not move away from the Linux ecosystem. Hence, it provides a less radical and on-demand approach toward DB-OS co-design. That is, you only specialize the subsystem that is the bottleneck of the DBMS. We summarize the strengths and weaknesses of the discussed co-designs in Table 5.1.

## 5.4 Case Study #1: High-Frequency Instantaneous VM Snapshot

In this section, we show how to specialize a virtual memory subsystem that can create/destroy snapshots orders of magnitude faster than Linux. Snappy targets database systems that split snapshotting into two roles. *Latency-sensitive foreground worker threads* process user-visible requests on the main page table; for example, Redis serves commands on its main event loop, and Hyper runs transaction processing on foreground workers. *Snapshot worker threads* operate on snapshot page tables to perform non-latency-sensitive work such as writing Redis RDB checkpoints or scanning a Hyper snapshot for analytical processing. As the analysis in Figure 5.2 has shown,

[1] https://github.com/cloudius-systems/osv/graphs/contributors?from=10%2F1%2F2022
[2] https://lwn.net/Articles/972605/

most of the CPU time is spent updating reference counts for every physical page. Therefore, one would naturally ask if removing reference counting during snapshot creation is possible. In Linux, this would be very hard as reference counting is a fundamental design technique leveraged by many features of the virtual memory subsystem. However, the key insight we draw is that in a privileged process, we can specialize in an extremely simple virtual memory subsystem, Snappy, just to serve the purposes of database snapshotting. Next, we show a lightweight scheme of creating, managing, and destroying snapshots at high frequency without using reference count.

### 5.4.1 Epoch-based Snapshotting

In Linux `fork`, the reference count of each physical page is to keep track of the number of references from PTEs in different page tables. A physical page can be returned to the system allocator only after its last reference has been dropped. Without per-page reference counting, we need a way to track these references for safe reclamation. First, we observe that in many database applications, the snapshot is discarded after use. This is the case for Hyper and Redis. Therefore Snappy makes a simplifying assumption: Snappy does not allow creating a snapshot from an existing snapshot. Next, we show how to efficiently track the references in a batch fashion, inspired by epoch-based reclamation [198,199].

The pseudo-code of our algorithms is listed in Figure 5.5. We associate each snapshot page table with an epoch number allocated from a monotonically increasing global timestamp when `dbos_snappy_spawn` loads a snapshot table copied from the main page table used by latency-sensitive foreground worker threads. A snapshot with epoch $E$ may contain any PTE copied before $E$ was assigned, so pages referenced by those PTEs cannot be reclaimed while that snapshot remains active. For each physical page X, we embed another epoch number `end_epoch`, assigned by reading the global counter after the link (PTE) from the main page table to X is removed.

**Reclaiming Physical Pages**. In our context, a physical page X can be reclaimed if it satisfies **(P1): there are no references to X from any page tables, including any cached PTEs in the TLB cache**. **P1** is met when a) X has been un-linked from the main page table, and b) there is no active snapshot whose epoch number is greater than or equal to X's `end_epoch`. Equivalently, b) can be expressed as `X.end_epoch < min_epoch(active snapshots)`. When X is unlinked from the main page table, we keep it in a global list (`limbo_list`) roughly ordered by `epoch_end`. Then, a background worker periodically reclaims physical limbo pages that satisfied **P1**.

**Synchronization**. To guard against concurrency anomalies, we use a read-write latch to serialize concurrent modifications to the main page table and snapshots and allow concurrent reads. For example, duplicating the main page table requires taking the read latch and modifying the snapshot set requires upgrading the read latch to write latch. For writes to the last level page table entry, we leverage compared-and-swap operations to reduce the need for a global exclusive latch. We defer the optimization for finer-grained latch (e.g., page) to future work.

### 5.4.2 Instantaneous Snapshot

While snapshot creation is significantly faster now as the random memory access has been drastically reduced, copying the page table is still on the critical path of the core creating the snapshot using the main page table. We can completely mask this latency from the critical path by copying the page table in advance asynchronously, making the snapshot appear instantaneous.

**Epoch-based Snapshot Algorithms**

```
E_global = 0 # global epoch
snapshots = {} # a set of snapshots in the system
limbo_list = [] # a list of physical pages to be reclaimed
def dbos_snappy_spawn(f: function): # Section 5.4.1
    dbos_make_snapshot()
    snapshot = snapshots.take()
    snapshot.epoch = E_global.add_and_fetch(1)
    snapshot.active = True
    disable writes for snapshotted regions in main page table
    run f on a CPU core with snapshot table

def dbos_snappy_make_snapshot(): # Section 5.4.1
    snapshot_table = make a copy of main page table
    disable writes for snapshotted address ranges
    snapshot_table.active = False # not active at creation
    snapshots.add(snapshot_table)

# Handlers registered at program start
def dbos_snappy_page_fault_handler(pgtbl, vaddr): # Section 5.4.1
    if write-protection fault and pgtbl is from snapshots:
        newp = copy page at vaddr
        update pgtbl at vaddr to point to newp
    elif write-protection fault and pgtbl is main page table:
        newp = copy page at vaddr
        page_meta(newp).end_epoch = max_epoch(active snapshots)
        limbo_list.add(newp)
        modify active snapshots to point to newp at vaddr
        make pte of pgtbl at vaddr writable  # Section 5.4.4
    tlb-shootdown on vaddr and synchronize main_pgtbl changes to inactive snapshots #
    ↪ Section 4.2/4.4

def dbos_snappy_syscall_handler(tf : Trapframe): # Section 5.4.3
    if using snapshot and syscall uses user-space buffers:
        Copy user-space buffers to/from staging buffer B
    Proxy syscall to the hypervisor/host kernel
```

Figure 5.5: **Algorithms for Epoch-based Snapshot Management**

To implement this, Snappy maintains a configurable number of page table copies from the main page tables in the background. PTEs of the snapshotted VM regions in the snapshot tables are disabled for writes so that they can be readily loaded by snapshot worker threads without doing further processing. When a snapshot is requested, user may retrieve one ready in the pre-copied set of snapshots. Notice we also replaced the main page table with a snapshot so that we do not have to disable writes for virtual pages in the main page table. To ensure that the snapshots are consistent, we synchronize any modifications to the main page table to the inactive snapshot tables inside the page fault handler as well as anywhere the main page table is updated. This way, the snapshot creation latency is completely hidden from the critical path. Furthermore, we can leverage multiple copy threads to keep enough inactive snapshots to serve a burst of snapshot requests.

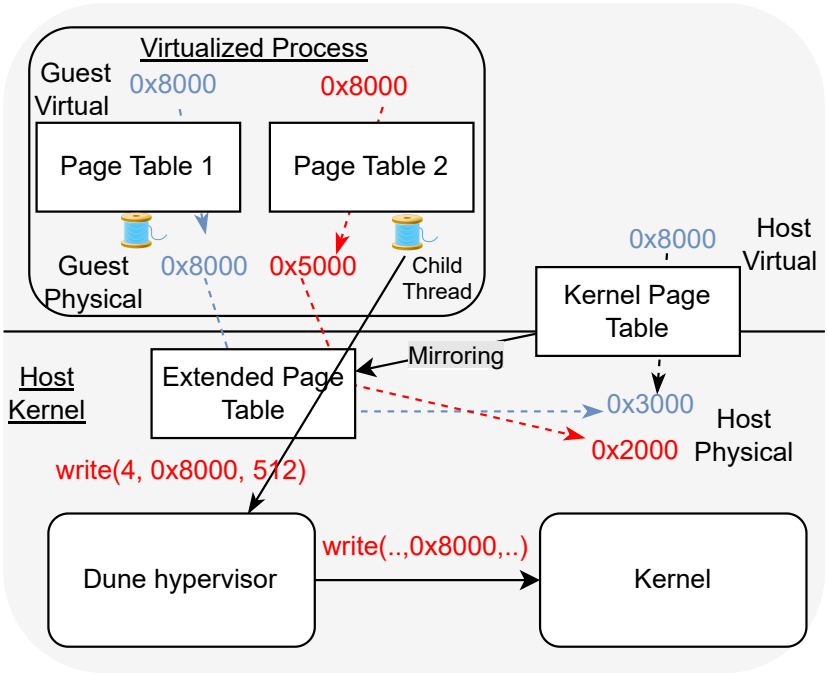



Figure 5.6: **Problems of Dune with Multiple Page Tables** - With one page table, Dune keeps a guest pointer and the numerically identical host pointer backed by the same host physical page. With multiple page tables, however, Page Table 2 can map GVA 0x8000 to a different guest physical page, and the EPT can map that guest physical page to HPA 0x2000. If the application passes pointer 0x8000 to `write`, the host kernel interprets it as HVA 0x8000 and follows the kernel page table to HPA 0x3000 instead.

### 5.4.3 Challenge: Supporting Multiple Page Tables

One unexpected challenge we faced when implementing such a VM system in a privileged process is supporting multiple page tables. One of Dune's fundamental design goals is to preserve the ability to issue ordinary system calls to the host operating system. To make that work, Dune relies on an address-translation invariant: when a guest thread passes a user pointer to the host

kernel, the host kernel must reach the same host physical page that the guest would reach by dereferencing that pointer.

Figure 5.6 shows why this invariant is easy to maintain with one guest page table. Consider the blue path. Inside the virtualized process, Page Table 1 translates GVA 0x8000 to guest physical address (GPA) 0x8000. The extended page table (EPT) then translates GPA 0x8000 to host physical address (HPA) 0x3000. The host kernel has its own page table for the same Linux process, and it maps host virtual address (HVA) 0x8000 to the same HPA 0x3000. Therefore, if the guest calls `write(fd, 0x8000, 512)`, the host kernel can simply treat 0x8000 as an ordinary process virtual address. Although the guest and host translations use different page tables, both translations end at HPA 0x3000, so the syscall reads the intended bytes.

The problem appears after Snappy introduces a second guest page table for a snapshot. The red path in Figure 5.6 shows a snapshot worker running on Page Table 2. Suppose it writes to GVA 0x8000 and triggers copy-on-write. The Libdbos fault handler allocates a new guest physical page, represented as GPA 0x5000, and updates only Page Table 2 so that GVA 0x8000 now maps to GPA 0x5000. When that guest physical page is first touched, the EPT maps GPA 0x5000 to a new host physical page, HPA 0x2000. From the snapshot worker's point of view, dereferencing GVA 0x8000 now correctly accesses HPA 0x2000.

However, the host kernel is unaware of which guest page table the calling core is using. If the snapshot worker calls `write(fd, 0x8000, 512)`, the numeric pointer passed across the syscall boundary is still just 0x8000. The host kernel interprets that value as HVA 0x8000 in the Linux process and follows the kernel page table, which still maps HVA 0x8000 to HPA 0x3000. As a result, the kernel writes the old page from Page Table 1's view instead of the copied page from Page Table 2's view. The root cause is that one guest virtual address can name different guest physical pages depending on the active guest page table, while a host system call has only one kernel page-table translation for that address.

To address this problem without modifying the application, Libdbos intercepts every system call that might access user-space memory. Libdbos internally keeps a staging buffer **B** whose mapping is shared consistently across all guest page tables and is not subject to CoW. For an output system call such as `write`, Libdbos first copies the bytes from the caller's current guest page-table view into **B**. It then passes the address of **B** to the hypervisor and, eventually, to the real host system call. Since every guest page table and the host kernel page table agree on the physical page backing **B**, the host kernel reads the same bytes that the caller intended to write. For input system calls such as `read`, Libdbos applies the same idea in the opposite direction: the host kernel writes into **B**, and Libdbos copies the returned bytes back into the caller's original user buffer using the caller's active guest page table. This general solution preserves application correctness transparently at the cost of an extra copy. We discuss possible optimizations in the next section.

### 5.4.4 Prioritized Copy-on-Write

The staging-buffer mechanism in the previous section is general, but it adds an extra copy to every system call that passes a user buffer. This overhead is most important for latency-sensitive foreground workers, because they serve user-visible requests on the main page table. Snappy therefore uses a specialized CoW policy that keeps the main page table compatible with the host kernel page table. We call this policy Prioritized-Copy-on-Write (PCoW).

The key idea is to decide which page table changes when a foreground worker writes to a protected page. In a conventional CoW implementation, the faulting thread receives a new physical page, so the fault handler changes the PTE in the page table that caused the fault. PCoW does the opposite for foreground workers. When a foreground worker using the main page table triggers a write-protection fault, LibDBOS keeps the main page-table entry unchanged and instead updates the corresponding PTEs in the snapshot page tables. Operationally, the snapshot workers are redirected to a copied page that preserves the snapshot's old contents, while the foreground worker continues to use the original page through the same main-page-table mapping.

This choice has two consequences. First, because PCoW does not change the main page table on foreground-worker writes, it avoids sending TLB shootdowns to all foreground-worker cores. At most, the faulting core needs to notify the snapshot worker cores whose snapshot page tables were changed, which is less disruptive because snapshot work is not on the request-latency path. Second, system calls from foreground workers no longer need the staging buffer. As shown in `dbos_snappy_page_fault_handler` in Figure 5.5, a foreground-worker CoW fault does not modify the faulting main-page-table PTE to a new guest physical page. Therefore, the main page table continues to agree with the host kernel page table, and the host kernel can safely interpret a foreground worker's user pointer directly. Snapshot workers still use the staging-buffer path described above, because their page tables may diverge from the host kernel page table.

### 5.4.5 Discussion

Implementing these mechanisms in Linux is theoretically possible but would require significant and invasive changes to its complex, security-critical memory subsystem, which consists of 101,000 lines of code[3]. For example, reference counting for physical pages is extensively used, with over 1,000 occurrences[4], spanning more than 300 source files. Modifying such a fundamental design principle would require a major overhaul of the Linux kernel. One attempt to reduce fork latency, on-demand-fork [178], applied copy-on-write (CoW) to the page table itself. However, this proposal faced resistance from kernel maintainers [200,201] due to concerns about potential instability and incompatibility. Another recent approach, async-fork [179], utilized asynchronous page table copying in a customized Linux kernel to accelerate `fork`. However, unlike our approach, which supports multiple snapshots within a short time window, async-fork allows only one asynchronous copy and must adhere to the reference-counting convention for physical pages, limiting its snapshot frequency to once per second. In contrast, with our proposed co-design, implementing these mechanisms becomes significantly easier because the LibDBOS guest kernel is drastically simplified. It delegates complex features—such as the page cache, shared memory, memory-mapped files, and swapping—to the host kernel. This allows for greater design flexibility within the LibDBOS kernel, free from the constraints of maintaining compatibility and security with the host kernel.

[3] Measured in the `mm` directory of Linux kernel v5.15.90

[4] Measured by counting the `get_page`/`put_page` invocations in the codebase

## 5.5 Case Study #2: Kernel Buffer Pool Without TLB-Shootdown

In this section, we turn our attention to TLB-shootdown, the general mechanism required in a traditional OS kernel, which causes scalability problems for buffer pool design. Specifically, we present an in-kernel buffer pool design, TABBY, that can fully utilize virtual memory hardware without incurring TLB shootdowns or high memory allocation overhead. This is uniquely enabled by our co-design paradigm, which allows the DBMS to access privileged instructions (TLB invalidation).

As Figure 5.7 shows, TABBY creates a virtual address range as large as the storage space. Initially, all the virtual pages have no mappings to physical pages. The mappings are gradually built through page faults, at which point physical pages are allocated and filled with data read from storage. TABBY fully controls the page fault handling for I/Os during page fault by hooking into the virtual memory subsystem of LIBDBOS. We next describe how TABBY eliminates TLB-shootdown when running in the kernel space.

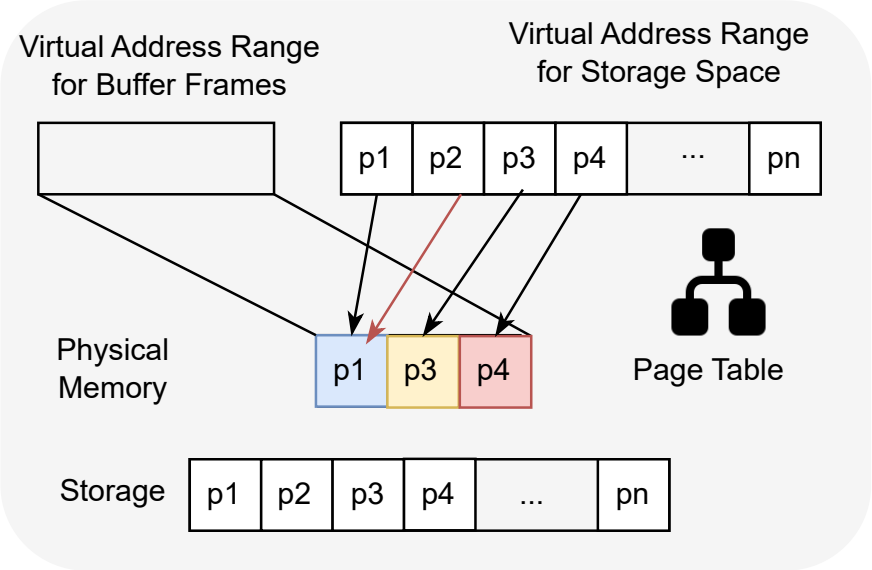



Figure 5.7: **TABBY Buffer Pool**

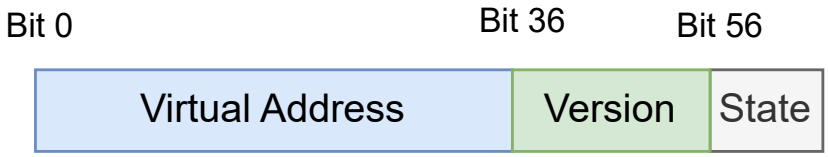



Figure 5.8: **TABBY Page Header Format**

### 5.5.1 Eliminating TLB-Shootdowns

In Linux, a TLB-shootdown is needed whenever a PTE is modified. For example, vmcache buffer manager design [22] uses `madvise` system call to remove a physical page from the page table,

**Tabby Buffer Pool Algorithms**

```
# vStorage : beginning virtual address for storage space
def dbos_tabby_pin_x(pid): # Pin page at pid exclusively
    off = pid * PageSize
    exp_addr = off + vStorage
    while True:
        PageHeader* h = vStorage + off
        PageHeader oldh = *h
        if oldh.vaddr != exp_addr: # possible stale mapping
            dbos_tabby_page_fault_handler(exp_addr)
            continue
        if oldh.state == Unlocked and h->CAS(oldh, PageHeader(exp_addr, oldh.version,
        ↪  Locked)):
            return vStorage + off

def dbos_tabby_unpin_x(pid):
    PageHeader* h = vStorage + pid * PageSize
    h->set_unlocked_bump_version()

def dbos_tabby_optimistc_read(pid, f):
    off = pid * PageSize
    exp_vaddr = off + vStorage
    while True:
        PageHeader* h = vStorage + off
        PageHeader oldh = *h # get a snapshot of the header
        if oldh.vaddr != exp_vaddr: # possible stale mapping
            dbos_tabby_page_fault_handler(exp_vaddr)
            continue
        if oldh.state == Locked:
            continue
        f(vStorage + off) # Read optimistically
        PageHeader oldh2 = *h # Read again for validation
        if oldh.version == oldh2.version and exp_addr == oldh2.vaddr and oldh2.state !=
        ↪  Locked: return
```

Figure 5.9: **Algorithms for Tabby Buffer Pool**

triggering an unscalable shootdown for every page eviction. This raises the question of whether it would be possible to leave the PTE (e.g., Page 2 in Figure 5.7) unchanged on eviction. If the PTE is not changed, then other cores' cached TLB entries for that virtual page remain consistent with the page table, so there is no immediate need to interrupt those cores and force them to invalidate the entry. However, later access to the virtual page might erroneously result in accessing the wrong physical page using the stale mappings. For security reasons, Linux disallows this – actively clearing the PTEs and performing a TLB-shootdown. In our approach, we do not need to worry about such security problems as the database is the only application and is isolated using virtualization. But we still have to reliably detect such stale mapping and fix them lazily. A key observation is that buffer pool pages are accessed through a well-defined `pin`/`unpin` API, which allows injecting explicit checks that detect stale mappings. To this end, Tabby embeds an 8-byte page header on each page, as shown in Figure 5.8. We next show how to use this small metadata

```
Tabby Page Fault Handling
def dbos_tabby_evict_if_needed():
    if number of free pages above threshold: return
    victims = pick unpinned and unlocked victims
    for pid in victims:
        PageHeader* h = vStorage + pid * PageSize
        PageHeader oldh = *h
        if oldh.state.dirty:
            # note the page is written with lock bit on
            lock page and write page to storage
        if oldh.state != Locked and h->CAS(oldh, InvalidPageHeader):
            free_page(vStorage + pid * PageSize)

# Page fault handler registered at program initialization
def dbos_tabby_page_fault_handler(addr):
    pte = pgtbl_lookup(addr)
    if pte->lock() == False: # One bit of PTE used for locking
        return # retry
    if pte->present == True: tlb_flush_one(addr)
    PageHeader* h = addr
    # Page is not present or the mapping might be stale
    if pte.present == False or h->vaddr != addr:
        dbos_tabby_evict_if_needed()
        new_page = alloc_page()
        read page from storage at off into new_page
        *pte = MakePTE(PA(new_page), Write+Present+Locked)
        # Page was locked when it was written to storage
        h->set_unlocked_bump_version()
    pte->unlock()
    tlb_flush_one(addr) # local tlb flush
```

Figure 5.10: **Page Fault Handling in Tabby**

to detect and fix stale mappings lazily without any TLB-shootdowns.

**Page Pinning**. We maintain the following invariant: *A page is only cached in at most one buffer frame uniquely identified by the virtual address in the header.* That is, for all the buffer frames containing valid pages, the virtual addresses in the headers all differ. As depicted in Figure 5.9, every pin operation first checks the virtual address of the page against the virtual address stored in the header (line 9). If they do not match, the mapping might be stale. Tabby handles this case as a page fault. Note that line 8 might also trigger a page fault due to a non-present page. If the address check passes, according to the invariant, the mapping is definitely not stale. Therefore, it proceeds to lock the page exclusively on the state bits in the header using compare and swap operations (Line 12). If the page is locked, it retries until the lock is acquired. Shared locking can be implemented similarly. For brevity, we omit its description.

**Page Fault Handling and Eviction**. As depicted in Figure 5.10, the page fault handler first looks up the PTE in the page table on the faulting address. Tabby serializes concurrent fault handling on the same address by locking on an unused bit in the PTE. Note that it is entirely possible that the page fault handler is called because of a stale TLB entry (Line 9/25 of Figure 5.9).

For example, suppose CPU core 1 has a stale TLB entry that maps Page 1 to the frame holding Page 3 instead. The address check will find that the virtual address in the header does not match Page 1's address (due to the invariant), even if the page table actually indicates Page 1 correctly points to Frame 1. This is possible due to previously cached stale TLB entries. Tabby handles this case by performing a cheap local TLB flush on the faulting address and rechecking the address (Line 18/21). The TLB flush ensures that access to the header (Line 21) uses the latest mapping in the page table.

After ensuring the mapping is truly stale or non-present, an I/O is needed to bring the page into the buffer pool. To make space in the buffer pool, Tabby first performs necessary replacement (`dbos_tabby_evict_if_needed`). The algorithm selects unpinned and unlocked victim pages using a clock replacement strategy. It then locks the victim pages and writes the pages to storage. Once the replacement is done, Tabby allocates a free buffer frame on which an I/O operation is performed to read the page into memory (Line 23-25). It then constructs a correct PTE. Note that the page read into the memory has the lock bit set when it was written back to storage (Line 25). Keeping the lock bit set during this write preserves the invariant. Finally, the PTE is unlocked, and a local TLB flush is performed to clear any stale TLB entries.

**Optimistic Read.** Similar to vmcache [22], Tabby supports optimistically reading a page without writing to shared memory for better scalability. As shown in `dbos_tabby_optimistc_read` in Figure 5.9, an optimistic read performs a cheap address check (Line 25) similar to pinning a page. Once the check passes, it performs the read if the page is not locked (Line 28-29). After the read, Tabby rereads the page header and validates that the version number is unchanged and that the page is still unlocked; this check detects concurrent writers that may have modified the page during the optimistic read. Unlike the optimistic read in vmcache, Tabby must also verify that the virtual address stored in the header still matches the requested page, because a stale TLB mapping could otherwise make the read observe a different buffer-frame page whose version number happens to be unchanged.

### 5.5.2 Physical Memory Allocation

The Linux kernel, designed to support multiple processes, also pays performance overhead for security. For example, when allocating a physical page for a non-present virtual page, the Linux kernel zeroes out the content of the allocated physical page before use for security reasons, as the page might have been used by other processes previously. Tabby, in contrast, can pre-allocate all the physical buffer frames once and reuse them repeatedly within the database process. Therefore, the allocator in Tabby does not require zeroing pages. To improve scalability, Tabby keeps free frames in a thread-local free list. When the local free list is exhausted, it requests a batch of free frames from a set of global lists, each protected by a latch.

### 5.5.3 Discussion

Tabby supports all the functionalities of vmcache without the overhead of TLB shootdowns and memory allocation. One might be tempted to implement such a lazy strategy on top of Linux in user-space. However, there are no Linux APIs for fixing the VM mapping without doing a TLB shootdown. `mremap` comes close to such semantics. However, `mremap` still requires a TLB-shootdown on the old address. Hence, it merely shifts the TLB-shootdown to a later time when

the evicted page is accessed again. To address the problems of vmcache, Leis et al. [22] proposed a Linux kernel module called exmap that specializes the virtual memory subsystem to address the kernel bottleneck by batching TLB-shootdowns. While exmap reduces the number of TLB shootdowns, each shootdown still requires every CPU core to completely flush its local TLB cache, negatively impacting the cache efficiency of modern CPUs. With modern storage devices and networks [194,202,203] capable of tens of millions of IOPS, the costs of TLB shootdowns remain significant, even with batching. In contrast, Tabby preserves cache efficiency by eliminating the need for shootdowns altogether. Beyond performance improvements, Tabby also has a smaller impact on host kernel security compared to exmap.

## 5.6 Libdbos Implementation

### 5.6.1 Libdbos Guest Kernel

We leveraged the open-source `libdune` [181] to implement this co-design paradigm as Libdbos. We made about 3000 lines of change. Besides basic OS utilities such as page table manipulation and system call proxy inherited from `libdune`, we added a scalable physical memory allocator with thread-local free lists and partitioned locks. We added support for sending inter-processor interrupts between vCPUs in the guest using posted interrupts, an Intel VT-d feature that allows a virtual CPU to receive interrupts without hypervisor intervention. We implemented a basic TLB-shootdown mechanism using IPI for basic OS functionality. When a vCPU intends to ask another vCPU to flush remote TLBs, the vCPU registers a function call to be executed on remote vCPU and then sends a IPI to the target vCPU. Note that this mechanism is not used in the Tabby buffer pool as it does not require TLB shootdowns. In the guest kernel, we intercept every system call made by the DBMS by setting `MSR_LSTAR` register (virtualized) to the address of a page storing system call entry code customized by Libdbos. We leverage this entry point to perform custom system-call interposition.

### 5.6.2 Libdbos Hypervisor

We built the hypervisor for Libdbos on top of Dune [181] for x86 architecture and made about 1250 lines of changes. We added a few enhancements to the Dune hypervisor:

**Host Huge Pages.** To reduce the two-dimensional translation overhead (TLB miss) in virtualization, we leverage huge pages of the host to back the physical memory of the process. This is common practice in modern hypervisor implementation [204].

**Guest Huge Pages.** Similarly, there is another layer of address translation inside the guest, which adds TLB-miss overhead. Libdbos provides guest huge page support to reduce the overhead of TLB miss. It intercepts `mmap` calls via its system call interposition capability and allocates huge guest pages for mapping greater than 2 MB on the guest page table. This helps reduce the page walk length inside the guest page table.

### 5.6.3 Snappy Implementation

We implemented the Snappy snapshot mechanism in approximately 1,200 lines of code within Libdbos. When Snappy is initialized, we replace the default page fault handler in Libdbos with

Snappy's custom page fault handling, as described in Figure 5.5. To handle system calls correctly (Section 5.4.3), we also override the default system call handler in Libdbos. With this custom system call handler, we can differentiate between system calls made by CPU cores using the main page table and those made on snapshots. For the latter, before proxying the system calls to the host kernel, we copy the user buffers associated with the system calls to and from a per-CPU staging buffer. To maintain synchronization during page table copying and updates, we utilize table-level locks.

### 5.6.4 Tabby Implementation

We developed Tabby by adapting the vmcache codebase[5]. When Tabby is initialized, we replace the default page fault handler with Tabby's custom page fault handling, as described in Figure 5.10. This required approximately 1,300 lines of modifications. We implemented the Clock replacement for buffer frames in Tabby, where each worker sequentially scans the physical buffer frames to identify eviction candidates. One bit of each page header is used to represent the reference bit required by the Clock algorithm. Unlike vmcache, which maintains a hash table to track resident buffer frames for eviction, Tabby stores buffer frames in a contiguous physical memory area backed by huge pages from the host. Tabby maps this contiguous physical memory to a continuous virtual memory area, allowing it to scan buffer frames without relying on additional data structures.

## 5.7 Experimental Evaluation

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

- What is the performance impact of elevated privilege levels?
- What are the latency benefits of the proposed snapshot mechanism compared to `fork` on an unmodified Linux kernel and an optimized `fork` mechanism using a specialized Linux kernel?
- What are the performance/efficiency benefits of Tabby buffer pool enabled by our co-design compared to the existing state-of-the-art buffer pools?

### 5.7.1 Baselines

**Snapshot Mechanism.**

We modified Redis (commit `bb524473`) to use the proposed snapshot system instead of using `fork` system call. Specifically, the modified Redis runs as a privileged process with the help of the hypervisor and Libdbos. It runs a background thread that operates on a snapshot page table to persist data in memory to storage. We then compare it against running Redis on an

[5] https://github.com/viktorleis/vmcache

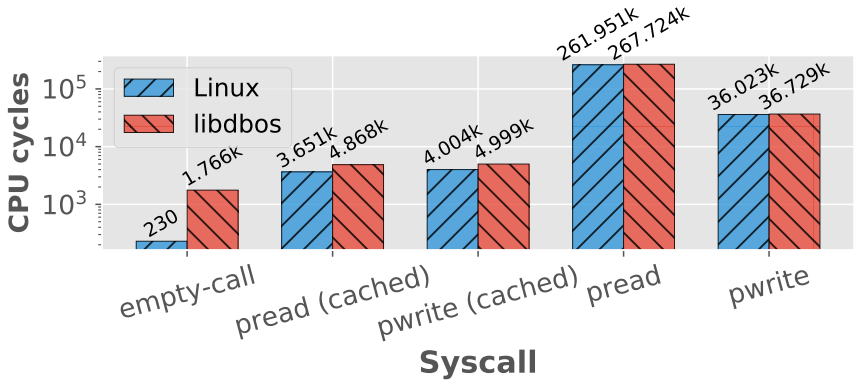



Figure 5.11: **Impact of Virtualization on System Calls**

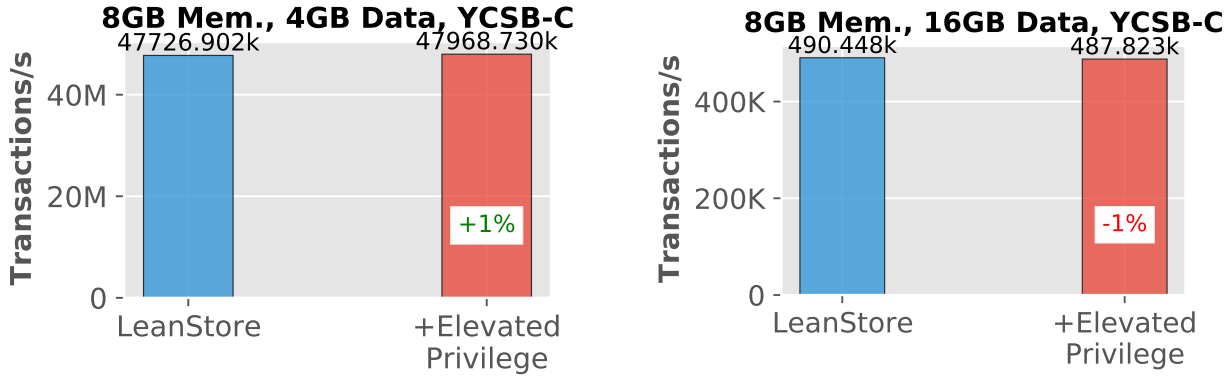



Figure 5.12: **Impact of Privilege Elevation on LeanStore**

unmodified Linux kernel (v6.6.1) and a modified Linux kernel (v6.0.6) [6] which implements On-Demand-Fork [178]. On-Demand-Fork applies copy-on-write to the page table itself so as to delay the page-table copy work as late as possible to remove the latency spike at `fork` time. We did not compare against async-fork [179] as we could not obtain the source code.

**Buffer Management.**

We compare Tabby against the following systems:

**vmcache**. vmcache is the state-of-the-art buffer management approach leveraging virtual-memory hardware directly for translation using POSIX `mmap` APIs. We use vmcache at commit `a8ed2b0`.

**LeanStore**. A buffer pool design that leverages pointer-swizzling for removing the hash table lookup overhead of the memory-resident pages. We use LeanStore at commit `dd42514` for evaluation. We configure LeanStore's isolation level to Read Uncommitted.

**WiredTiger**. A traditional buffer pool design that employs software hash table for page table lookup. We use WiredTiger v2.6.1 for evaluation. We configure WiredTiger to use the lowest isolation level, Snapshot Isolation, implemented with MVCC.

[6] https://github.com/rssys/on-demand-fork

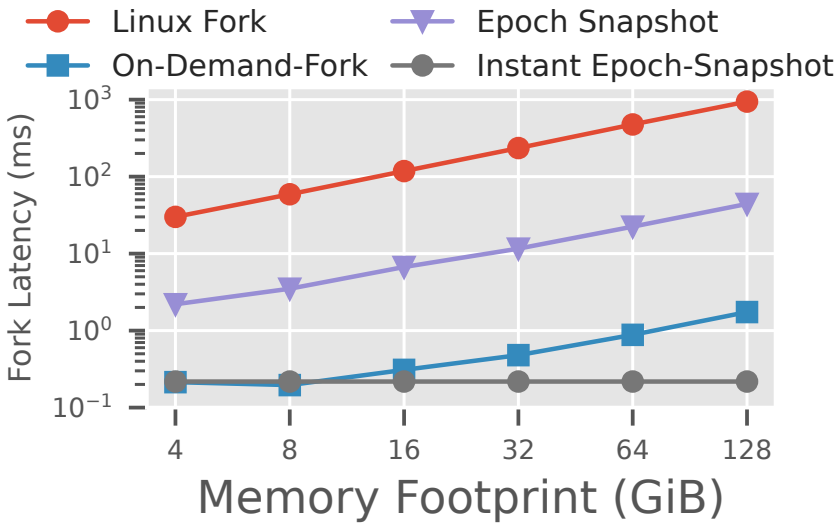
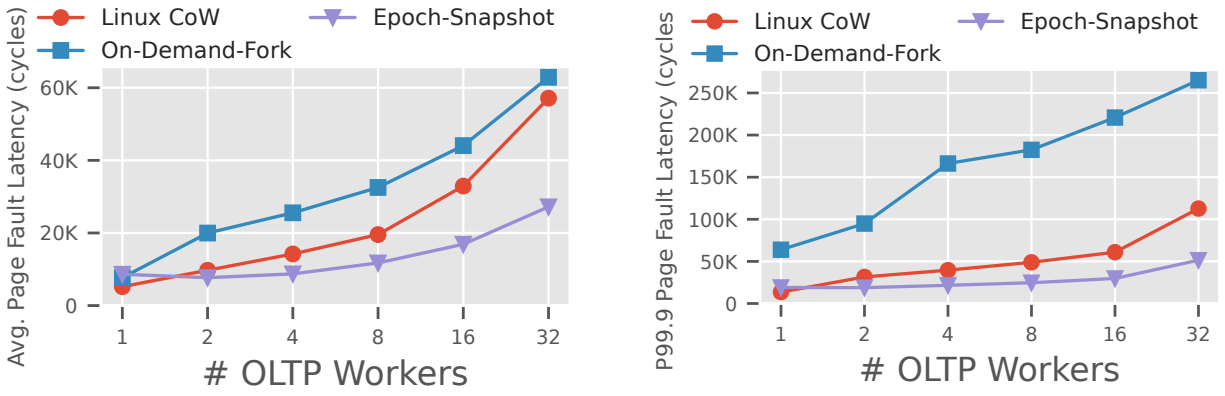



Figure 5.13: **HTAP Micro-Benchmark** - Snapshot Latency




(a) **Average Fault Latency**

(b) **P99.9 Fault Latency**

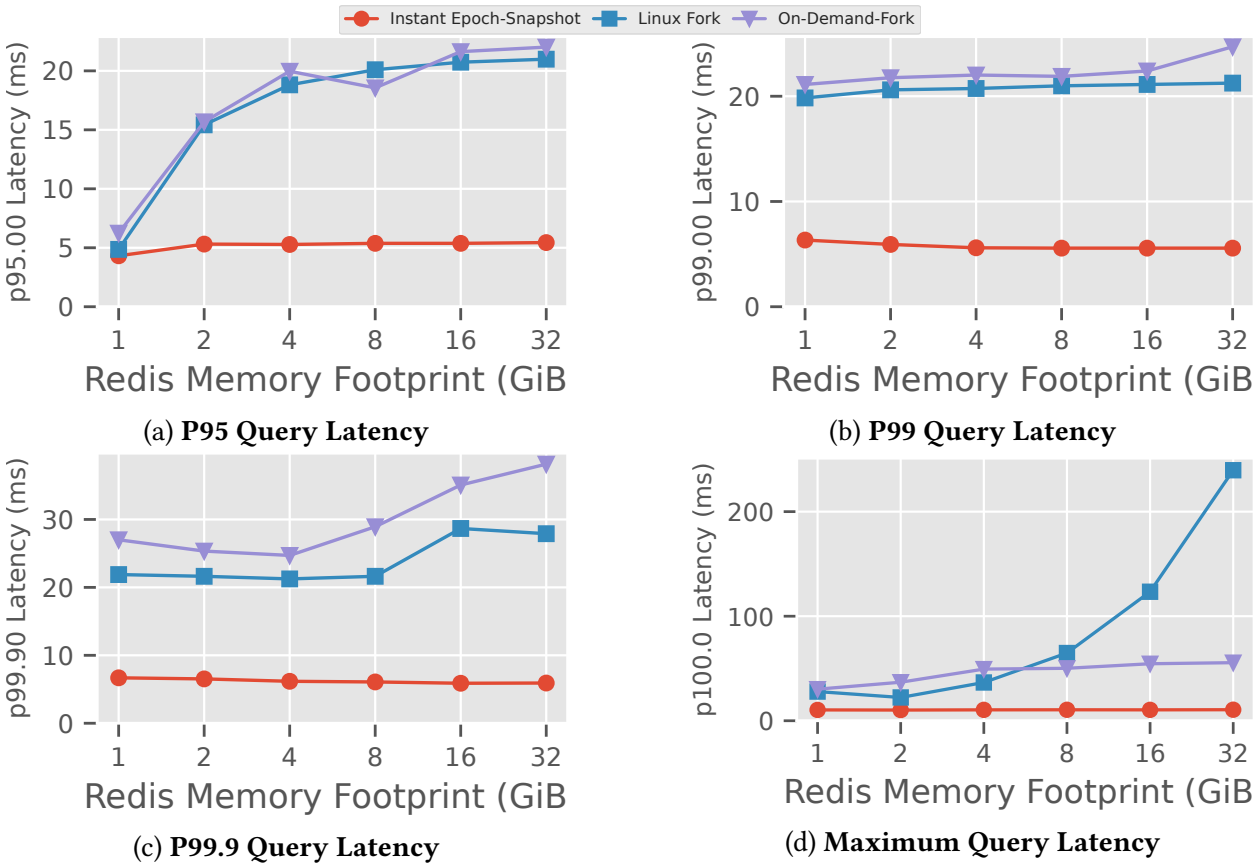
Figure 5.14: **HTAP Micro-Benchmark** - OLTP Latency after Snapshot

### 5.7.2 Experimental Setup

All experiments are conducted on a server with two Intel(R) Xeon(R) Gold 5320 CPU @ 2.20GHz with 104 cores and 755 GB DRAM. The server is running Linux kernel 6.6.1 as the operating system. We use a 3.84 TB NVMe SSD (Intel P5500 RI U.2). For buffer pool experiments, we configure all systems to use direct IO and disable write-ahead logging and the Linux page cache for all experiments.

### 5.7.3 Impact of Privilege Elevation

We begin by analyzing the performance impact of virtualization and privilege elevation on system calls, with a focus on I/O operations such as `pread` and `pwrite` that are used by most DBMSes. For comparison, we also measured the latency of an empty system call (`gettid`), representing the worst-case scenario for a privileged DB process. The results, as shown in Figure 5.11, indicate that



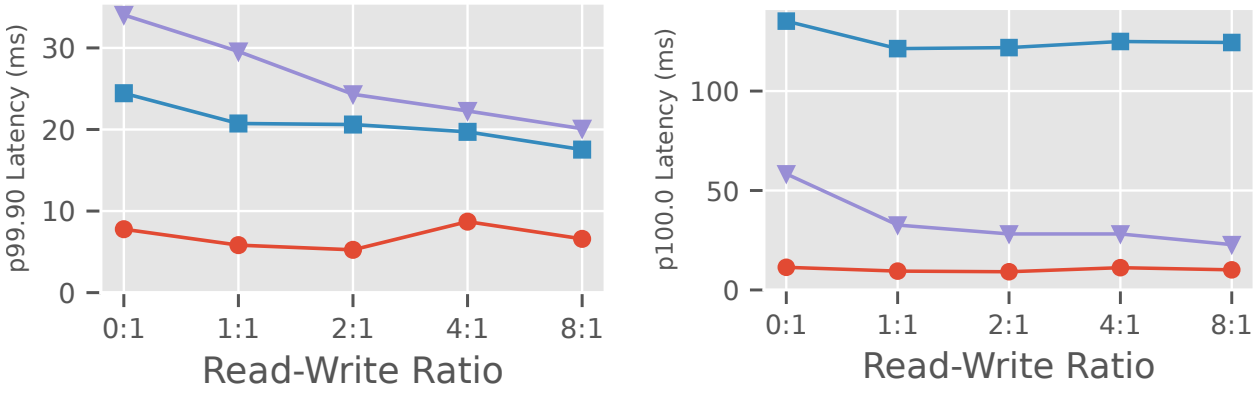

(a) **P95 Query Latency**

(b) **P99 Query Latency**

(c) **P99.9 Query Latency**

(d) **Maximum Query Latency**

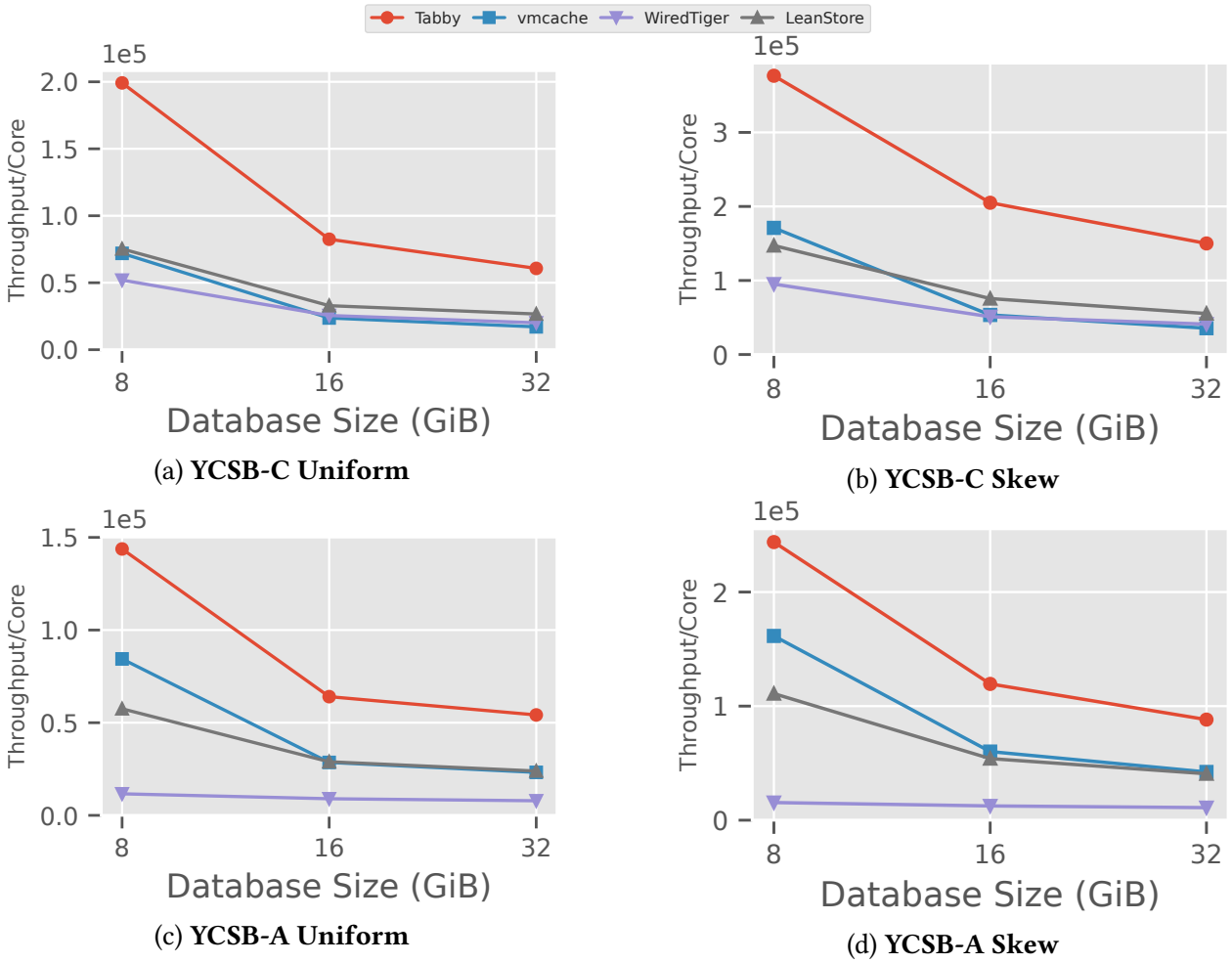
Figure 5.15: **Tail Latency of 3M Write Queries as Database Size Varies (Queue Depth=1000, Payload Size=1 KB, Key Pattern=Parallel)** - We trigger a snapshot(**BGSAVE** command) during the benchmark.




(a) **P99.9 Query Latency**

(b) **Maximum Query Latency**

Figure 5.16: **Tail latency of 3M write queries against a 16 GB Redis database varying read-write ratios (Queue Depth=1000, Payload Size=1 KB, Key Pattern=Gaussian) after snapshot**




(a) **YCSB-C Uniform**

(b) **YCSB-C Skew**

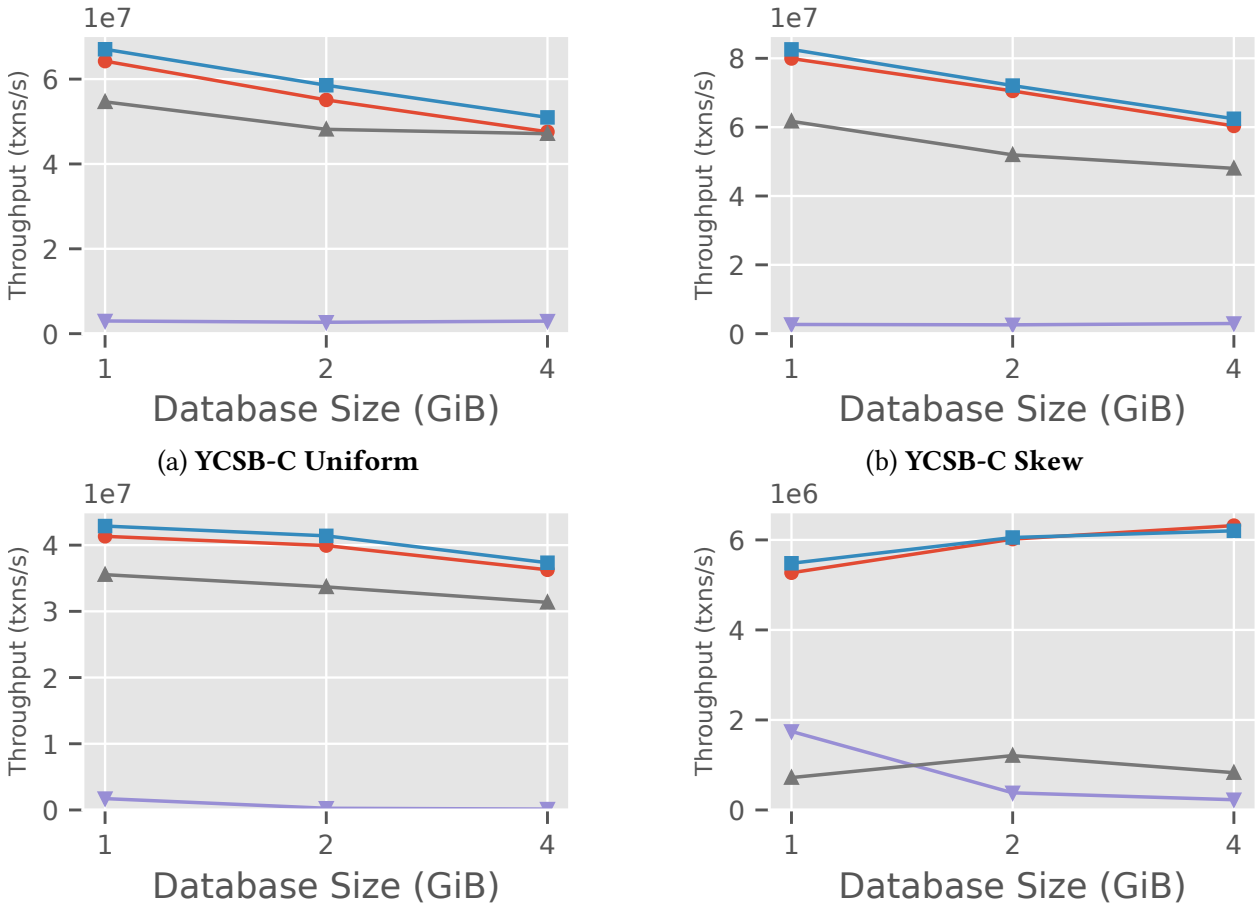
(c) **YCSB-A Uniform**

(d) **YCSB-A Skew**

Figure 5.17: **Throughput/Core on out-of-memory workloads** - (8GB Buffer Pool, 64 Threads). We derive this metric by normalizing absolute throughput by the effective number of cores required to achieve the absolute throughput.

virtualization imposes a fixed overhead of 1,536 CPU cycles, mainly used in exiting and entering the virtualization boundary for proxying system call. For I/O system calls, the performance impact is much less than for an empty system call such as `gettid`. For example, for uncached `pread` and `pwrite` calls, the performance impact is minimal as the storage medium latency dominates the overhead. In contrast, when data is cached in the Linux page cache, the overhead rises to approximately 30% for `pread` and 24% for `pwrite`. These findings demonstrate that the privileged DB process imposes reasonable performance overhead on system calls, particularly those dominated by storage latency.

We also study whether running a storage engine in a privileged space (Guest Kernel Space) has a positive performance impact. We ran the LeanStore buffer pool in a privileged process and compared it against vanilla LeanStore on Linux with YCSB-C workloads and 64 threads. The results are shown in Figure 5.12. When data fits in memory, we did not observe a measurable throughput difference. When the buffer pool cannot cache all data, privileged LeanStore performs only 1% worse than LeanStore running in the user space of Linux due to VM exit/entry. Hence,




(a) **YCSB-C Uniform** (b) **YCSB-C Skew**

(c) **YCSB-A Uniform** (d) **YCSB-A Skew**

Figure 5.18: **Throughput of different buffer pool designs on in-memory workloads** - (8GB Buffer Pool, 64 Threads)

without judiciously leveraging the hardware constructs available, running at a higher privilege level does not translate to higher performance.

### 5.7.4 Snapshot Mechanism Evaluation

#### HTAP Micro-benchmark

In this section, we evaluate the proposed snapshot mechanisms using a set of micro-benchmarks that simulate HTAP workloads. We use multiple threads to randomly write to a 4 GB array to simulate concurrent OLTP workloads. For each snapshot taken, we run a thread to sequentially scan the array to simulate a typical OLAP workload. Note that we enable prioritized copy-on-write (PCoW) mechanism discussed in Section 5.4.4.

**Snapshot Latency**. We first measure the latency of creating a snapshot (i.e., fork) by an OLTP thread, varying the memory footprint. Illustrated in Figure 5.13, our epoch-based snapshot improves latency by up to 21× compared to Linux `fork`, thanks to removing reference counting. Note that `on-demand-fork` has even lower snapshot latency as it defers copying the page table

to the future (see next paragraph for the cost). By adding asynchronous copying, denoted by `Instant Epoch-Snapshot`, we achieve similar or lower latency than `on-demand-fork`. Note that we assume that the time interval between two snapshot requests is smaller than the time it takes to copy the page table. We believe this assumption is reasonable as Epoch-Snapshot creates a 128 GB memory snapshot in 44ms with a single core, which can be further parallelized.

**OLTP Latency.** To understand the impact of snapshot and CoW on OLTP threads, we measure the latency to resolve a write-protection fault. We show the results in Figure 5.14a and Figure 5.14b. Generally, `Epoch-Snapshot` generally outperforms Linux and `on-demand-fork` as the number of OLTP threads increases by up to 2.1×/2.3× for the average latency and 2.3×/7.3× for 99.9-percentile latency. Thanks to the PCoW optimization, the number of processors that must participate in a TLB shootdown is reduced compared to Linux. `on-demand-fork` has significantly higher tail latency because deferred page table copying is carried out on CoW fault, causing 512 random memory updates to reference counters. Interestingly, with 1 OLTP thread, Epoch-Snapshot has a 40% higher page fault latency than Linux CoW. This is because Linux does not require TLB-shootdown when a process has only one thread. With PCoW, we still need to send a TLB shootdown to the snapshot core because it is unaware of the change in its page table.

**OLAP Throughput**. For brevity, we omit results on OLAP throughput after snapshot as they were similar on all baselines.

### Redis Workload

Next, we evaluate the impact of snapshot and CoW mechanisms on latency when applied to a real-world in-memory database system, Redis. For a performance metric, we focus on tail latency, including maximum latency, as they are important for in-memory caches and databases for user-facing services [205–207].

**Uniform Workload.** We first run an experiment studying the write query latency after a snapshot is taken while varying the database size. We use `memtier_benchmark` [7] to generate workload traffic. We issue a total of $3\times10^6$ queries to a pre-populated and pre-warmed Redis server while keeping a query queue depth (i.e., number of outstanding queries) to around 1000. During the benchmark, we trigger the snapshot by sending one `BGSAVE` command to Redis. We record the latency of every query and plot the results in Figure 5.15. We can see that `Instant Epoch-Snapshot`, compared to unmodified Linux, significantly reduces latency across all the measured percentile points by as much as 15× on maximum latency, achieving the lowest tail latency. Surprisingly, `on-demand-fork` is only effective for the maximum latency (Figure 5.15d) and reduces the latency by 4×. For lower percentiles, `on-demand-fork` is sometimes outperformed by an unmodified Linux Kernel. This is because of its deferred page table copying overhead on CoW faults, resulting in higher latency for these write queries. `Instant Epoch-Snapshot`, on the other hand, performs these page-table copying work in advance and does not impose additional processing overhead on normal CoW faults on data pages.

**Skewed Workload.** Real-world workloads exhibit skewed patterns in the keyspace. Therefore, we configure the `memtier_benchmark` tool to generate keys that conform to a Gaussian distribution. We pre-populate the Redis server to host $16\times10^6$ key-value pairs, each with 1 KB in size. The memory footprint is about 16 GB. We then issue $3\times10^6$ queries against Redis while

[7] https://github.com/RedisLabs/memtier_benchmark

varying the read-write ratio in the workload. The results are shown in Figure 5.16. Linux and `on-demand-fork` tail latency decreases as there are more reads in the workload, which results in fewer page faults overall. `Instant Epoch-Snapshot` has the lowest P99.9 latency curve, outperforming Linux and `on-demand-fork` by 2.6×/3.6×. `on-demand-fork` performs worse than Linux on P99.9 latency for similar reasons explained in the previous section. `on-demand-fork` and `Instant Epoch-Snapshot` both outperform Linux on maximum latency, as the snapshot call impacts this metric the most.

To summarize, `Instant Epoch-Snapshot` significantly reduces the tail latency for in-memory systems that leverage virtual memory for snapshotting compared to using `fork`.

### 5.7.5 Buffer Pool Evaluation

**YCSB Benchmark.** In this experiment, we use YCSB-C and YCSB-A as the main workloads to evaluate buffer pool designs. We use a key-value pair consisting of an 8-byte key and 120 random bytes of payload. We evaluated our designs mainly with Uniform and Zipfian [208] access distribution. By default, we use a Zipfian skew factor of 0.9 where 80% of the accesses land on ∼10% of the keys. We ran the workloads with 64 worker threads for a warm-up phase of 1-minute after loading. We report average throughput of a 2-minute workload phase after warm-up. We configure vmcache and Tabby to use all 64 threads for request processing and eviction. LeanStore uses a dedicated pool of threads for eviction. We tuned and configured 8 Page Provider threads and 56 worker threads for LeanStore.

**Out-of-Memory Workloads.** In out-of-memory scenarios, all systems must repeatedly evict pages and bring new pages from storage, so raw throughput alone does not capture how much CPU work is required to sustain that throughput. We therefore normalize throughput by the effective number of CPU cores used during the experiment, as shown in Figure 5.17. Under this metric, Tabby achieves up to 3.5× higher throughput per core than vmcache and up to 2.5× higher throughput per core than LeanStore. The bottlenecks are different for the two baselines. vmcache loses efficiency because eviction relies on Linux virtual-memory operations that trigger TLB shootdowns; in our profiling, vmcache spends about 70% of CPU cycles in the kernel on TLB-shootdown work. LeanStore does not suffer from this kernel TLB-shootdown path, but its dedicated Page Provider threads consume CPU cycles scanning for eviction candidates and maintaining enough free frames for worker threads. In contrast, Tabby avoids kernel TLB-shootdown overhead and performs replacement on demand in worker threads, so it does not need a separate eviction-thread pool. The same trend holds for both uniform and skewed out-of-memory workloads.

**In-Memory Workloads.** When data fits in memory, Tabby and vmcache perform similarly on read-only workloads ( Figure 5.18a and Figure 5.18b) because they leverage hardware translation and optimistic read. LeanStore comes next due to having fewer worker threads. WiredTiger suffers from both contention and hash table lookup overhead. On write-heavy uniform workloads ( Figure 5.18c), we observe similar patterns. However, when the access pattern is skewed ( Figure 5.18d), all systems experience write contention and throughput drops significantly, while LeanStore suffers the most. The contention causes frequent system calls for suspending threads due to read-write lock contention. Tabby and vmcache suffer the least because they leverage spinlock and rarely enter the kernel.

In summary, Tabby achieves state-of-the-art performance when data fits in memory and is much more resource-efficient on out-of-memory workloads than competitors.

**Impact of TLB-shootdown on Cache Efficiency.** Next, we examine the performance impact of TLB shootdowns on CPU cache efficiency when the virtual memory (VM) subsystem is frequently updated. We simulate scenarios in which some worker threads operate entirely in memory on skewed data, while others frequently update VM mappings (using exmap) for I/O operations. The results, shown in Figure 5.19, demonstrate that TLB shootdowns can have up to a 2× impact on in-memory threads, even though these threads are independent of the I/O threads. This impact arises from the cycles spent trapping into the interrupt handler for TLB flushes, which lead to TLB cache misses. However, this issue does not affect Tabby, as there are no TLB shootdowns when manipulating page tables due to our tighter co-design.

**TPC-C**. We next evaluate the buffer pool systems using TPC-C which is a write-heavy workload with complex access patterns, including range scans and deletes. We configure a 16 GB buffer pool for all the systems. We load 512 warehouses into the database before the benchmark, which amounts to ∼100 GB on storage. We use 64 threads to run the workload. We run the workload for 2 minutes and report the average throughput. The results are shown in Figure 5.20. Tabby, vmcache, and LeanStore are all able to saturate the SSD, achieving similar throughput. They outperform WiredTiger by about 7×. In terms of per-core-throughput, shown in Figure 5.20b, Tabby consistently outperforms vmcache and LeanStore by about 2×. Compared to WiredTiger, Tabby achieves 4.2× higher throughput per core.

**Performance Drill-down.** We next break down the impact of various techniques used in Tabby. We start with vmcache running on Linux as the baseline and incrementally enable techniques. The results are shown in Figure 5.21. Surprisingly, when running vmcache with elevated privilege level, we observe 60% lower efficiency. This is due to the TLB-shootdown overhead being doubled because Libdbos hypervisor needs to perform an additional TLB-shootdown operation per `madvise` system call on the PTEs of the extended page table (EPT). When TLB-shootdown is eliminated with Tabby algorithms, the efficiency increases by 753%, outperforming the baseline by 3.5×. Thread-local free lists and no-zeroing optimizations add another 8% and 4% improvements. Hence, we can conclude that TLB-shootdown elimination is the most important optimization, especially because privileged execution can otherwise add another layer of shootdown overhead.

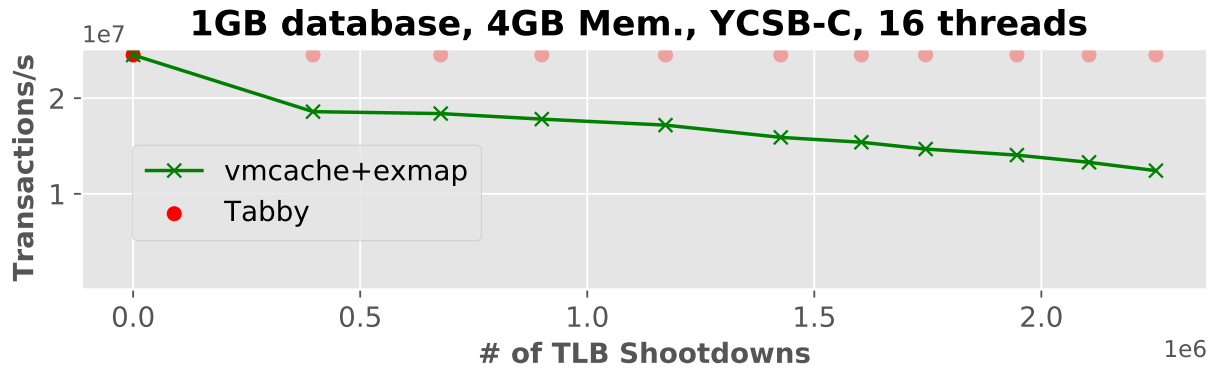
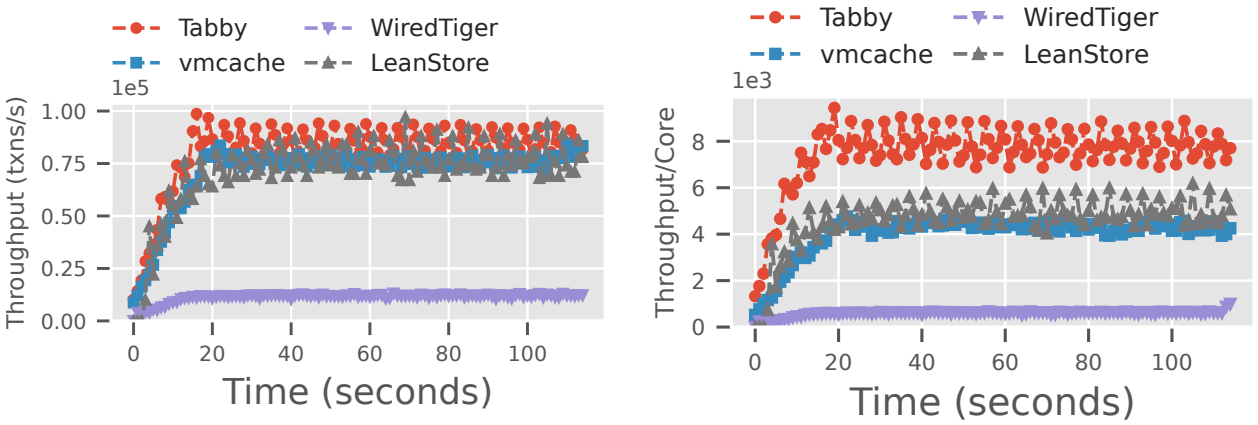



Figure 5.19: **Performance Impact of TLB-shootdown on Cache Efficiency of In-Memory Workloads for vmcache+exmap**




(a) **Throughput over Time**

(b) **Throughput/Core over Time**

Figure 5.20: **TPC-C Throughput over Time (16 GB Buffer Pool, 64 Threads, 512 Warehouses≈100 GB)**

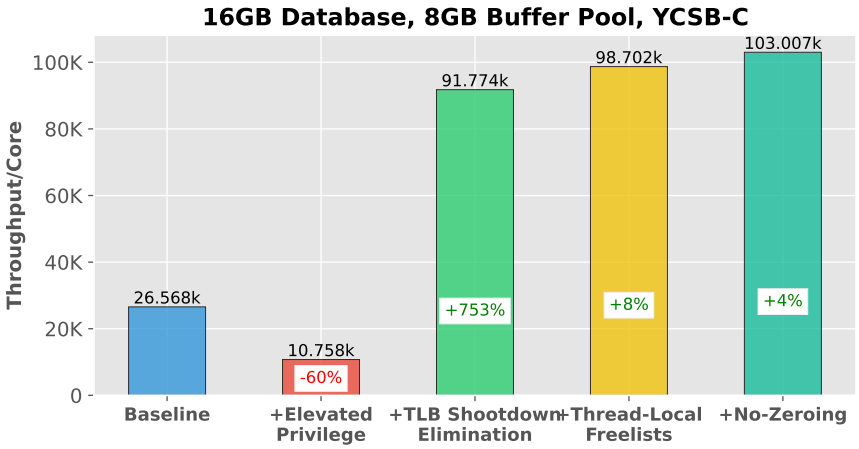



Figure 5.21: **Performance Drill-down on Tabby**

## 5.8 Related Work

While Section 5.2 focused on the OS and virtualization mechanisms that directly motivate LIBDBOS, this section discusses the broader body of work on DB-OS co-design, kernel bypass, virtual memory, snapshotting, and buffer management.

**DB-OS Co-Design.** The DBOS project [209] focuses on building a cluster OS using scalable distributed databases [28,210–212], for better resource management, observability, serverless application development [213], and debugging [214]. Currently, the DBOS project still runs the DBMS in user space on Linux. Our co-design approach allows the DBMS in the DBOS project more control over the hardware. MxKernel [215] is a runtime system that advocates run-to-completion execution rather than OS threads for database systems. COD [216] studies how to better integrate DBMS with OS for better co-optimization regarding resource management. Our

co-design approach focuses more on practically empowering DBMS with more abstractions and privileged instructions. Therefore, our work is complementary to these studies.

## 5.9 Summary

In conclusion, we present a novel DB-OS co-design paradigm leveraging the privileged DB process, unlocking new abstractions that are only possible with privileged instructions while minimizing the impact on security, maintainability, compatibility, and ecosystem. We presented two DB-OS co-design mechanisms for data-intensive applications enabled by this paradigm, including an in-kernel buffer pool design without TLB-shootdown overhead and a high-frequency instantaneous snapshot mechanism for in-memory databases and HTAP workloads. Our proposed mechanisms demonstrated significant performance and efficiency improvements, as well as tail-latency reductions relative to existing approaches based on kernel modification. Privileged kernel bypass also opens additional uses beyond snapshotting and buffer management, which we discuss as future work in the thesis conclusion.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

This thesis began with the DB–OS co-design space shown in Figure 1.1. The goal of this thesis was to show how database systems can remove OS boundary overhead without abandoning the compatibility, security, and deployability that production DBMSes require.

The first step was to establish why this problem matters on modern hardware. We revisited the classic OLTP bottleneck analysis on a memory-optimized engine and showed that communication and OS-kernel overhead now dominate CPU time across YCSB, Voter, and TPC-C, even when transactions are executed as stored procedures (Chapter 2). These results reframed the OS boundary as the high pole in the tent for in-memory database performance, and they set the agenda for the rest of the thesis.

The second step addressed communication without giving up the kernel ecosystem. We built Tux, a partial kernel-bypass networking stack on top of the existing eBPF and XDP fast path (Chapter 3). Tux pairs a message-based transport protocol with a pushdown abstraction that runs DBMS logic directly on network cores through a preemptible user-space runtime. Because it reuses kernel NIC drivers, it deploys without user-space drivers or exclusive NIC ownership. Across VoltDB, Redis, Memcached, and ScyllaDB, Tux matches or beats production-grade DPDK-based stacks on throughput and tail latency without giving up the compatibility that those stacks sacrifice.

The third step revisited the long-standing trade-off between OS-managed and user-space buffer management. We built Calico, a DBMS-controlled buffer pool whose multi-level array-based address translation, together with path caching, hole punching, and group prefetching, gets the best of both worlds: in-memory speed comparable to `mmap`-style OS translation and out-of-memory speed comparable to traditional hash-table-based user-space buffer pools, without forcing the database to choose between them (Chapter 4). Calico excels across a diverse mix of workloads, including sequential scans, B-tree lookups, and graph-based vector search. We integrated Calico into PostgreSQL with modest engineering effort.

The final step addressed mechanisms that user space cannot reach. We built Libdbos, a framework that uses hardware virtualization to run the database in a privileged guest while leaving the host kernel binary untouched (Chapter 5). The two case studies on top of Libdbos, Snappy for instantaneous VM-level snapshots and Tabby for TLB-shootdown-free buffer management, illus-

trate what becomes possible once the database can execute privileged instructions and manipulate page tables directly. Together, these four steps form a practical recipe for removing OS-boundary overhead from a database while preserving the compatibility, security, and deployability that production database systems demand.

## 6.2 Future Work

We briefly discuss several directions to extend the work presented in this thesis.

One direction is to push Tux beyond the OLTP and key-value workloads it was designed for. Distributed analytical query processing exercises the network stack in a very different way. A single query may shuffle hundreds of gigabytes of intermediate results across nodes in a short window, and the dominant cost shifts from per-request latency to sustained data-plane throughput. Extending Tux's protocol and pushdown API to carry large, often columnar payloads efficiently is a natural next step. The shift in workload also exposes a part of the design space that the OLTP evaluation in this thesis did not stress. Tux was evaluated on OLTP traffic that was small enough to keep the network well below saturation, and the experiments saw essentially no congestion or packet loss. Analytical queries with data shuffles invalidate that assumption, and the resulting incast and buffer pressure on the NIC and the switch become concerns. Generic end-to-end congestion control reacts only to loss or delay, with no awareness of which exchange a packet belongs to or how much data the executor still intends to send. A promising direction is to co-design Tux's transport and flow control with the query executor so that congestion decisions can use information the executor already has. The pushdown runtime is also a natural place to host streaming exchange operators such as repartitioning, hash-join build-side staging, and partial aggregation on the network cores so that intermediate data is shaped before it reaches the worker pool. In a disaggregated cloud database, page fetches, log shipping, and remote scan results all flow through the same network fabric that an analytical query uses for shuffling, and a single Tux-style stack with high-throughput message transport, query-aware congestion control, and programmable pushdown could serve as a common substrate for both the storage and the compute layers.

Another direction is to follow new hardware trends that are reshaping the boundary between the database and the rest of the system. SmartNICs [217] and DPUs [218] expose programmable cores directly on the data path, and they are a natural target for the Tux stack. The pushdown abstraction and the database-assisted scheduling that selects between inline execution and worker dispatch map cleanly onto the on-NIC cores, so offloading Tux to a SmartNIC would free host cores for query execution while preserving the same programming model the DBMS already targets, and it would let the network stack live on the NIC. CXL-attached memory and persistent memory introduce new design considerations for the database architecture, expanding the memory hierarchy with new tiers whose latency, bandwidth, and persistence characteristics differ enough that the OS page cache cannot manage them well on the database's behalf. Calico is a natural starting point: its multi-level array-based translation gives fast translation that scales across very large address spaces, which is exactly what a tiered buffer pool spanning DRAM, CXL memory, persistent memory, and SSD requires. The translation layer would stay where it is, but eviction, promotion, and migration between tiers would move under DBMS control rather than being delegated to the OS, so that the database can place hot pages in DRAM, keep warm pages in CXL

or persistent memory, and use access patterns to drive migration policy.

Another direction is to extend the same partial-bypass approach as Tux to the NVMe SSD storage stack, which this thesis did not address. Today the choice on the storage path resembles the network path before Tux: either go through the lengthy kernel storage stack, including caching and filesystem layers, and pay significant overheads, or give the DBMS the entire NVMe device using a kernel-bypass library such as SPDK and lose the filesystem abstraction along with everything that depends on it. A natural question is whether one can reach SPDK-class performance without giving up the kernel filesystem. One promising path is to let the DBMS issue targeted NVMe commands directly to the device only for its hot files, while the kernel filesystem continues to manage namespace, allocation, and metadata. eBPF on the storage side, in the spirit of XRP [219], gives a plausible mechanism for safely splicing a database-aware fast path into an in-tree storage stack. A key challenge is that the database's own buffer pool may hold a more recent version of a page than what is on the device, so the fast path cannot simply bypass the DBMS cache. Resolving this requires co-design between the storage stack and the DBMS so that direct device reads either consult the buffer pool first or are restricted to pages the DBMS guarantees are clean on disk.

Privileged kernel bypass also opens directions beyond the two Libdbos mechanisms studied in this thesis. One example is faster memory rewiring for query processing and indexing. Memory rewiring remaps existing virtual-memory regions instead of copying data, and prior work has used it to accelerate query operators and index structures [167,220–223]. On Linux, these techniques typically rely on `mmap`, which makes remapping expensive because the kernel must maintain general-purpose process metadata, including per-page reference counts, for the affected physical pages. The virtual-memory techniques behind Snappy suggest a more specialized implementation point. A DBMS running in a privileged process could maintain the page-table state needed for remapping without inheriting all Unix process-management costs, making memory-rewiring techniques more practical for database operators and data structures. Privileged kernel bypass could also be combined with existing storage and networking bypass mechanisms. Recent systems use eBPF and related mechanisms to move caching, proxying, and other data-intensive functionality closer to the kernel fast path [46–48,68,180]. These mechanisms preserve much of the Linux ecosystem, but they remain constrained by the kernel's safety model and do not directly expose privileged CPU and virtual-memory mechanisms to the DBMS. A privileged DB process is complementary because it can still use system calls and kernel extension points for deployable I/O paths, while specializing page translation, snapshotting, memory rewiring, and privilege transitions inside the database process.

Another opportunity is to use privileged kernel bypass as a substrate for stronger isolation and scheduling inside the DBMS. Many DBMSes support stored procedures, UDFs, extensions, or loadable modules that run inside the DBMS address space for performance and interoperability. This design is efficient, but it is risky in shared or multi-tenant settings because faulty or malicious code can access DBMS memory directly. Chapter 2 showed that stronger isolation mechanisms, such as process or VM isolation, can make communication overhead due to isolation completely dominate execution time. A privileged DB process suggests a different split in which the DBMS keeps privileged control over its own virtual-memory subsystem, while user-defined logic runs in a less privileged protection domain. This could provide memory isolation for user-defined code without forcing every interaction through heavyweight process or VM boundaries. The same privilege boundary could support lightweight preemptive threading. User-space threading libraries

are efficient, but they cannot directly program interrupt hardware for preemption. Linux kernel threads provide preemption, but the kernel scheduler is general-purpose and can be expensive for fine-grained database scheduling [224]. A privileged DB process could use interrupt hardware and system-call interception to build a DBMS-specific preemptive scheduler with lower overhead, helping database engines expose enough concurrency to saturate high-end I/O devices [193] while still letting the DBMS schedule according to transaction priorities, I/O waits, and internal contention.

Finally, the design principles that emerged from this thesis, reusing non-critical-path OS functionality, building DB-native mechanisms for the critical path, and using virtualization to cross the privilege boundary safely, are not specific to the systems built here. As database workloads continue to diversify into vector search, machine learning, and graph analytics, and as hardware continues to evolve toward smarter NICs and persistent memory, the surface area where the database meets the operating system will keep growing. The work in this thesis is one step toward making that surface a practical interface through which databases can extract modern hardware performance without paying the practicality costs of earlier kernel-bypass designs.

# References


[1] International Data Corporation. *Worldwide IDC Global DataSphere Forecast, 2024–2028: AI Everywhere, But Upsurge in Data Will Take Time*. https://www.marketresearch.com/IDC-v2477/Worldwide-IDC-Global-DataSphere-Forecast-36971010/. 2024.

[2] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. "Retrieval-augmented generation for knowledge-intensive NLP tasks." In: *NeurIPS*. 2020.

[3] Y. A. Malkov and D. A. Yashunin. "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs." *IEEE transactions on pattern analysis and machine intelligence*, **42**(4), 2018, pp. 824–836.

[4] H. Jegou, M. Douze, and C. Schmid. "Product quantization for nearest neighbor search." *IEEE transactions on pattern analysis and machine intelligence*, **33**(1), 2010, pp. 117–128.

[5] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi. "Diskann: Fast accurate billion-point nearest neighbor search on a single node." *Advances in neural information processing Systems*, **32**, 2019.

[6] *Vantage: Cloud Cost Observability and Optimization*. https://instances.vantage.sh/.

[7] T. Steinert, M. Kuschewski, and V. Leis. "Cloudspecs: Cloud Hardware Evolution Through the Looking Glass." In: *Conference on Innovative Data Systems Research (CIDR)*. 2026. URL: https://www.cidrdb.org/cidr2026/papers/p17-steinert.pdf.

[8] M. Jasny, M. El-Hindi, T. Ziegler, and C. Binnig. "A Wake-Up Call for Kernel-Bypass on Modern Hardware." In: *Proceedings of the 21st International Workshop on Data Management on New Hardware*. 2025, pp. 1–5.

[9] X. Zhou, V. Leis, X. Yu, and M. Stonebraker. "OLTP Through the Looking Glass 16 Years Later: Communication is the New Bottleneck." In: *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, Netherlands*. 2025.

[10] M. Stonebraker. "Operating system support for database management." *Communications of the ACM*, **24**(7), 1981, pp. 412–418.

[11] Wikipedia. *Transaction Processing Facility*. URL: https://en.wikipedia.org/wiki/Transaction_Processing_Facility.

[12] Wikipedia. *Pick Operating System*. URL: https://en.wikipedia.org/wiki/Pick_operating_system.

[13] *Data Plane Development Kit*. URL: https://www.dpdk.org/.

[14] 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)*. 2014, pp. 489–502.

[15] 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*. 2019, pp. 1–16.

[16] *F-Stack*. URL: https://github.com/F-Stack/f-stack.

[17] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, et al. "The demikernel datapath os architecture for microsecond-scale datacenter systems." In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 195–211.

[18] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. "SPDK: A development kit to build high performance storage applications." In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2017, pp. 154–161.

[19] A. Madhavapeddy and D. J. Scott. "Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework?" *Queue*, **11**(11), 2013, pp. 30–44.

[20] V. Leis and C. Dietrich. "Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware." *PVLDB*, **17**(8), 2024, pp. 2115–2122.

[21] A. Sharma, F. M. Schuhknecht, and J. Dittrich. "Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting." In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 245–258.

[22] V. Leis, A. Alhomssi, T. Ziegler, Y. Loeck, and C. Dietrich. "Virtual-Memory Assisted Buffer Management." *Proceedings of the ACM on Management of Data*, **1**(1), 2023, pp. 1–25.

[23] K. Lim, M. Yoon, K. Kim, A. Fekete, and H. Jung. "Rapid Data Ingestion through DB-OS Co-design." *Proc. ACM Manag. Data*, **3**(1), 2025. DOI: 10.1145/3709718. URL: https://doi.org/10.1145/3709718.

[24] A. Crotty, V. Leis, and A. Pavlo. "Are you sure you want to use mmap in your database management system." In: *CIDR 2022, Conference on Innovative Data Systems Research. https://db. cs. cmu. edu/papers/2022/p13-crotty. pdf*. 2022.

[25] *PostgreSQL*. https://www.postgresql.org/.

[26] *MySQL*. http://mysql.com.

[27] A. Kemper and T. Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In: *ICDE*. ICDE. 2011, pp. 195–206. ISBN: 978-1-4244-8959-6.

[28] R. Kallman et al. "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System." In: *VLDB*. 2008, pp. 1496–1499.

[29] M. Stonebraker and A. Weisberg. "The VoltDB Main Memory DBMS." *IEEE Data Eng. Bull.*, **36**(2), 2013, pp. 21–27.

[30] A. Kemper and T. Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pp. 195–206.

[31] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. "LeanStore: In-memory data management beyond main memory." In: *ICDE*. IEEE. 2018, pp. 185–196.

[32] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. "Speedy Transactions in Multicore In-memory Databases." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. 2013, pp. 18–32.

[33] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. "The SAP HANA Database–An Architecture Overview." *IEEE Data Eng. Bull.*, **35**(1), 2012, pp. 28–33.

[34] *Growing Pains - DPDKs move to API/ABI Stability and its effect on OVS*. URL: https://www.youtube.com/watch?v=UyIRf7QccKQ.

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

[36] M. A. Cusack, J. Adamson, M. Brinicombe, N. A. Carson, T. Kejser, J. Peterson, A. Vasudev, K. Westerfeld, and R. Wipfel. "Yellowbrick: An Elastic Data Warehouse on Kubernetes." In: *CIDR*. 2024.

[37] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. "OLTP through the Looking Glass, and What We Found There." In: *SIGMOD*. 2008.

[38] A. Pavlo. "What are we doing with our lives? Nobody cares about our concurrency control research." In: *SIGMOD*. 2017.

[39] Oracle. *Main Components of Oracle JVM*. https://docs.oracle.com/en/database/oracle/oracle-database/19/jjdev/Oracle-JVM-components.html.

[40] D. Buchanan. *A library to assist writing memory-unsafe code in "pure" python, without any imports*. https://github.com/DavidBuchanan314/unsafe-python.

[41] M. H. Jim Keniston Prasanna S Panchamukhi. *Kernel Probes (Kprobes)*. https://docs.kernel.org/trace/kprobes.html.

[42] S. Dronamraju. *Uprobe-tracer: Uprobe-based Event Tracing*. https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt.

[43] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. "Amazon aurora: Design considerations for high throughput cloud-native relational databases." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1041–1052.

[44] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, et al. "Socrates: The new sql server in the cloud." In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1743–1756.

[45] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff. "Revisiting the open vswitch dataplane ten years later." In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021, pp. 245–257.

[46] M. Butrovich, K. Ramanathan, J. Rollinson, W. S. Lim, W. Zhang, J. Sherry, and A. Pavlo. "Tigger: A database proxy that bounces with user-bypass." *PVLDB*, **16**(11), 2023, pp. 3335–3348.

[47] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller. "{BMC}: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing." In: *NSDI*. 2021, pp. 487–501.

[48] M. Butrovich. "On Embedding Database Management System Logic in Operating Systems via Restricted Programming Environments." PhD thesis. Carnegie Mellon University, 2024.

[49] L. Zhu, Y. Shen, E. Xu, B. Shi, T. Fu, S. Ma, S. Chen, Z. Wang, H. Wu, X. Liao, et al. "Deploying user-space {TCP} at cloud scale with {LUNA}." In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 2023, pp. 673–687.

[50] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. "{IX}: a protected dataplane operating system for high throughput and low latency." In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 2014, pp. 49–65.

[51] V. Srinivasan, B. Bulkowski, W.-L. Chu, S. Sayyaparaju, A. Gooding, R. Iyer, A. Shinde, and T. Lopatic. "Aerospike: Architecture of a real-time operational dbms." *Proceedings of the VLDB Endowment*, **9**(13), 2016, pp. 1389–1400.

[52] Z. Yang, C. Yang, F. Han, M. Zhuang, B. Yang, Z. Yang, X. Cheng, Y. Zhao, W. Shi, H. Xi, et al. "OceanBase: a 707 million tpmC distributed relational database system." *Proceedings of the VLDB Endowment*, **15**(12), 2022, pp. 3385–3397.

[53] *ScyllaDB*. URL: https://docs.scylladb.com/manual/stable/kb/dpdk-hardware.html.

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

[55] 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."

[56] *MySQL X Protocol*. https://dev.mysql.com/doc/dev/mysql-server/8.4.3/mysqlx_protocol_implementation.html.

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

[58] *Redis serialization protocol specification*. https://redis.io/docs/latest/develop/reference/protocol-spec/.

[59] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. "Homa: A receiver-driven low-latency transport protocol using network priorities." In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 221–235.

[60] J. Ousterhout. "A linux kernel implementation of the homa transport protocol." In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 99–115.

[61] L. Shalev, H. Ayoub, N. Bshara, and E. Sabbag. "A cloud-optimized transport protocol for elastic and scalable hpc." *IEEE micro*, **40**(6), 2020, pp. 67–73.

[62] T. Ziegler, D. Bindiganavile Mohan, V. Leis, and C. Binnig. "Efa: A viable alternative to rdma over infiniband for dbmss?" In: *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 2022, pp. 1–5.

[63] *A Remote Direct Memory Access Protocol Specification*. URL: https://datatracker.ietf.org/doc/html/rfc5040.

[64] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. "Snap: A microkernel approach to host networking." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 399–413.

[65] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. "Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads." In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 361–378.

[66] J. Fried, G. I. Chaudhry, E. Saurez, E. Choukse, Í. Goiri, S. Elnikety, R. Fonseca, and A. Belay. "Making kernel bypass practical for the cloud with junction." In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 55–73.

[67] *Kernel page-table isolation*. URL: https://en.wikipedia.org/wiki/Kernel_page-table_isolation.

[68] Y. Zhou, X. Xiang, M. Kiley, S. Dharanipragada, and M. Yu. "{DINT}: Fast {In-Kernel} Distributed Transactions with {eBPF}." In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 401–417.

[69] M. Butrovich, S. Arch, W. Shen Lim, W. Zhang, J. M. Patel, and A. Pavlo. "BPF-DB: A Kernel-Embedded Transactional Database Management System For eBPF Applications." *To appear at SIGMOD 2025*, 2025.

[70] 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*. 2018, pp. 54–66.

[71] *Stream Control Transmission Protocol*. URL: https://datatracker.ietf.org/doc/html/rfc4960.

[72] *Stream Control Transmission Protocol*. URL: https://en.wikipedia.org/wiki/Stream_Control_Transmission_Protocol.

[73] *Setting up a xsk*. URL: https://docs.ebpf.io/linux/concepts/af_xdp/%5C#setting-up-a-xsk.

[74] *Goroutines*. URL: https://go.dev/doc/effective_go#goroutines.

[75] *Coroutines*. URL: https://www.iso.org/standard/73008.html.

[76] Y. He, J. Lu, and T. Wang. "CoroBase: coroutine-oriented main-memory database engine." *arXiv preprint arXiv:2010.15981*, 2020.

[77] R. Iyer, M. Unal, M. Kogias, and G. Candea. "Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling." In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 466–481.

[78] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. "Characterizing, Modeling, and Benchmarking {RocksDB}{Key-Value} Workloads at Facebook." In: *FAST*. 2020, pp. 209–223.

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

[80] *Deploying eBPF, XDP & AF_XDP for Cloud Native*. URL: https://archive.fosdem.org/2021/schedule/event/sdn_ebpf_afxdp/.

[81] *Placement groups for your Amazon EC2 instances*. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[82] *Max MTU of XDP programs*. URL: https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_XDP/#__tabbed_1_1.

[83] *NoSQL Redis and Memcache traffic generation and benchmarking tool*. URL: https://github.com/RedisLabs/memtier_benchmark.

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

[85] *ScyllaDB Shard-per-Core Architecture*. URL: https://www.scylladb.com/product/technology/shard-per-core-architecture/.

[86] *Cassandra Stress*. https://github.com/scylladb/cassandra-stress.

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

[88] X. Zhou, V. Leis, J. Hu, X. Yu, and M. Stonebraker. "Practical db-os co-design with privileged kernel bypass." *Proceedings of the ACM on Management of Data*, **3**(1), 2025, pp. 1–27.

[89] V. Leis and C. Dietrich. "Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware." *PVLDB*, **17**, 2024.

[90] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki. "Interleaving with coroutines: a practical approach for robust index joins." *Proceedings of the VLDB Endowment*, **11**(2), 2017, pp. 230–242.

[91] C. Jonathan, U. F. Minhas, J. Hunter, J. Levandoski, and G. Nishanov. "Exploiting coroutines to attack the" killer nanoseconds"." *Proceedings of the VLDB Endowment*, **11**(11), 2018, pp. 1702–1714.

[92] K. Huang, T. Wang, Q. Zhou, and Q. Meng. "The art of latency hiding in modern database engines." *Proceedings of the VLDB Endowment*, **17**(3), 2023, pp. 577–590.

[93] H. Liu, R. Li, Z. Zhang, and B. Tang. "Tao: Improving Resource Utilization while Guaranteeing SLO in Multi-tenant Relational Database-as-a-Service." *Proceedings of the ACM on Management of Data*, **2**(4), 2024, pp. 1–26.

[94] J. Mühlig and J. Teubner. "MxTasks: How to Make Efficient Synchronization and Prefetching Easy." In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 1331–1344.

[95] K. Huang, J. Zhou, Z. Zhao, D. Xie, and T. Wang. "Low-Latency Transaction Scheduling via Userspace Interrupts." 2025.

[96] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. "Arrakis: The Operating System Is the Control Plane." *ACM Trans. Comput. Syst.*, **33**(4), Nov. 2015. ISSN: 0734-2071. DOI: 10.1145/2812806. URL: https://doi.org/10.1145/2812806.

[97] D. Peng, C. Liu, T. Palit, A. Vahldiek-Oberwagner, M. Vij, and P. Fonseca. "Pegasus: Transparent and Unified Kernel-Bypass Networking for Fast Local and Remote Communication." In: *Proceedings of the Twentieth European Conference on Computer Systems*. 2025, pp. 360–378.

[98] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. "{MICA}: A holistic approach to fast {In-Memory}{Key-Value} storage." In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 429–444.

[99] A. Kalia, M. Kaminsky, and D. Andersen. "Datacenter {RPCs} can be general and fast." In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 1–16.

[100] A. Sanaee, V. Jabrayilov, I. Marinos, A. Kalia, D. Saxena, P. Goyal, K. Kaffes, and G. Antichi. "Fast Userspace Networking for the Rest of Us." *arXiv preprint arXiv:2502.09281*, 2025.

[101] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. "Shinjuku: Preemptive Scheduling for {$\mu$second-scale} Tail Latency." In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 345–360.

[102] Y. Li, N. Lazarev, D. Koufaty, T. Yin, A. Anderson, Z. Zhang, G. E. Suh, K. Kaffes, and C. Delimitrou. "Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling." In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2024, pp. 922–936.

[103] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. "Caladan: Mitigating interference at microsecond timescales." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 281–297.

[104] W. Effelsberg and T. Haerder. "Principles of database buffer management." *ACM Transactions on Database Systems (TODS)*, **9**(4), 1984, pp. 560–595.

[105] T. Neumann and M. J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance." In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[106] M. Zinsmeister, L.-D. Nguyen, V. Leis, and T. Neumann. "Predictive Translation: High-Performance Buffer Management Without the Trade-Offs." *Proc. ACM Manag. Data*, **4**(1), Apr. 2026. DOI: 10.1145/3786678. URL: https://doi.org/10.1145/3786678.

[107] pgvector contributors. *pgvector hnsw graph traversal*. https://github.com/pgvector/pgvector/blob/master/src/hnswutils.c#L818. 2025.

[108] sqlite-vec contributors. *A vector search SQLite extension that runs anywhere!* https://github.com/asg017/sqlite-vec. 2025.

[109] Y. Zhang, S. Liu, and J. Wang. "Are there fundamental limitations in supporting vector data management in relational databases? A case study of PostgreSQL." In: *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE. 2024, pp. 3640–3653.

[110] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang. "Spann: Highly-efficient billion-scale approximate nearest neighborhood search." *Advances in Neural Information Processing Systems*, **34**, 2021, pp. 5199–5212.

[111] L. Kuffo, E. Krippner, and P. Boncz. "PDX: A data layout for vector similarity search." *Proceedings of the ACM on Management of Data*, **3**(3), 2025, pp. 1–26.

[112] C. Fu, C. Xiang, C. Wang, and D. Cai. "Fast approximate nearest neighbor search with the navigating spreading-out graph." *arXiv preprint arXiv:1707.00143*, 2017.

[113] W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton. "The oracle universal server buffer manager." In: *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE). 1997, pp. 590–594.

[114] SQLite Development Team. *sqlite*. https://github.com/sqlite/sqlite/blob/master/src/pcache1.c. 2025.

[115] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch. "In-Memory Performance for Big Data." *VLDB*, **8**(1), 2014.

[116] Jonathan Corbet. *The final step for huge-page swapping*. https://lwn.net/Articles/758677/. 2018.

[117] *madvise Linux manual page*. https://man7.org/linux/man-pages/man2/madvise.2.html. 2024.

[118] *madvise MADVDONTNEED implementation in Linux kernel*. https://elixir.bootlin.com/linux/v6.17.7/source/mm/madvise.c. 2024.

[119] Hironobu SUZUKI. *PostgreSQL Buffer Tag*. https://www.interdb.jp/pg/pgsql08/01.html. 2024.

[120] Oracle. *Extent Descriptor Page of InnoDB*. https://dev.mysql.com/blog-archive/extent-descriptor-page-of-innodb/. 2024.

[121] *sys.dm_os_buffer_descriptors (Transact-SQL)*. https://learn.microsoft.com/en-us/sql/relational-databases/system-dynamic-management-views/sys-dm-os-buffer-descriptors-transact-sql?view=sql-server-ver17. 2024.

[122] *RocksDB's BlockCache key format*. https://github.com/facebook/rocksdb/blob/main/cache/cache_key.cc. 2024.

[123] *Bigfile Tablespaces in Oracle Database*. https://docs.oracle.com/en/database/oracle/oracle-database/26/admin/managing-tablespaces.html#GUID-7E376766-D99D-4274-BF00-8DC1E6FC5063. 2024.

[124] *A library for efficient similarity search and clustering of dense vectors*. https://github.com/facebookresearch/faiss. 2024.

[125] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. "Sharing buffer pool memory in multi-tenant relational database-as-a-service." *Proceedings of the VLDB Endowment*, **8**(7), 2015, pp. 726–737.

[126] *WiredTiger's source tree*. https://github.com/wiredtiger/wiredtiger. 2024.

[127] R. Stoica and A. Ailamaki. "Enabling Efficient OS Paging for Main-Memory OLTP Databases." In: *DaMon*. 2013.

[128] G. Henry. "Howard chu on lightning memory-mapped database." *Ieee Software*, **36**(06), 2019, pp. 83–87.

[129] RavenDB. *RavenDB's source tree*. https://docs.ravendb.net/4.0/server/storage/storage-engine/. 2025.

[130] T. Michailidis, A. Delis, and M. Roussopoulos. "Mega: Overcoming traditional problems with os huge page management." In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. 2019, pp. 121–131.

[131] A. Manocha, Z. Yan, E. Tureci, J. L. Aragón, D. Nellans, and M. Martonosi. "Architectural support for optimizing huge page selection within the OS." In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 2023, pp. 1213–1226.

[132] S. J. White. *Pointer swizzling techniques for object-oriented database systems*. The University of Wisconsin-Madison, 1994.

[133] R. Otaki, J. H. Chang, A. J. Elmore, and G. Graefe. "Enhancing Transaction Processing through Indirection Skipping." *Proc. VLDB Endow.*, **18**(11), Sept. 2025, pp. 4104–4116. ISSN: 2150-8097. DOI: 10.14778/3749646.3749680. URL: https://doi.org/10.14778/3749646.3749680.

[134] Microsoft. *SQLServer Index architecture and design guide*. https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver17#index-placement-on-filegroups-or-partitions-schemes. 2025.

[135] Oracle. *Oracle Index Storage*. https://docs.oracle.com/en/database/oracle/oracle-database/26/cncpt/indexes-and-index-organized-tables.html#GUID-832C2B66-912B-4E1C-B4B5-AA40F49E4970. 2025.

[136] PostgreSQL Global Development Group. *PostgreSQL Database File Layout*. https://www.postgresql.org/docs/current/storage-file-layout.html. 2025.

[137] PostgreSQL Global Development Group. *PostgreSQL btree implementation*. https://github.com/postgres/postgres/blob/master/src/backend/access/nbtree/nbtsearch.c#L107. 2025.

[138] Oracle. *MySQL btree implementation*. https://github.com/mysql/mysql-server/blob/trunk/storage/innobase/btr/btr0cur.cc#L883. 2025.

[139] Microsoft. *VirtualAlloc function (memoryapi.h)*. https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc. 2025.

[140] LWN.net. *Introducing support for huge zero pages*. https://lwn.net/Articles/1033058/. 2025.

[141] F. Mahling, M. Weisgut, and T. Rabl. "Fetch Me If You Can: Evaluating CPU Cache Prefetching and Its Reliability on High Latency Memory." In: *Proceedings of the 21st International Workshop on Data Management on New Hardware*. 2025, pp. 1–9.

[142] *Memory-level parallelism :: Apple M2 vs Apple M4*. https://lemire.me/blog/2025/07/09/memory-level-parallelism-apple-m2-vs-apple-m4/. 2025.

[143] *Measuring the memory-level parallelism of a system using a small C++ program*. https://lemire.me/blog/2018/11/05/measuring-the-memory-level-parallelism-of-a-system-using-a-small-c-program/. 2018.

[144] T. David, R. Guerraoui, and V. Trigonakis. "Everything you always wanted to know about synchronization but were afraid to ask." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 33–48.

[145] P. E. McKenney. "Memory barriers: a hardware view for software hackers." *Linux Technology Center, IBM Beaverton*, 2010.

[146] A. Vardanian. *USearch by Unum Cloud*. Version 2.21.3. Oct. 2023. DOI: 10.5281/zenodo.7949416. URL: https://github.com/unum-cloud/usearch.

[147] Yandex Research. *Benchmarks for Billion-Scale Similarity Search*. https://research.yandex.com/blog/benchmarks-for-billion-scale-similarity-search. 2024.

[148] *Datasets for approximate nearest neighbor search*. http://corpus-texmex.irisa.fr. 2024.

[149] KShivendu. *dbpedia-entities-openai-1M*. https://huggingface.co/datasets/KShivendu/dbpedia-entities-openai-1M. 2026.

[150] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. "The end of an architectural era: (it's time for a complete rewrite)." In: *VLDB*. 2007, pp. 1150–1160. ISBN: 978-1-59593-649-3.

[151] 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: *SIGMOD*. 2013, pp. 1243–1254.

[152] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. "Anti-caching: A New Approach to Database Management System Architecture." *Proc. VLDB Endow.*, **6**(14), Sept. 2013, pp. 1942–1953.

[153] A. Eldawy, J. Levandoski, and P.-Å. Larson. "Trekking through siberia: Managing cold data in a memory-optimized database." *PVLDB*, **7**(11), 2014, pp. 931–942.

[154] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes." In: *SIGMOD*. 2016, pp. 1567–1581.

[155] A. van Renen et. al. "Managing Non-Volatile Memory in Database Systems." In: *SIGMOD*. 2018.

[156] X. Zhou, J. Arulraj, A. Pavlo, and D. Cohen. "Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory." In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 2195–2207.

[157] N. Riekenbrauck, M. Weisgut, D. Lindner, and T. Rabl. "A three-tier buffer manager integrating CXL device memory for database systems." In: *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*. IEEE. 2024, pp. 395–401.

[158] X. Hao, X. Zhou, X. Yu, and M. Stonebraker. "Towards buffer management with tiered main memory." *Proceedings of the ACM on Management of Data*, **2**(1), 2024, pp. 1–26.

[159] T. Ziegler, C. Binnig, and V. Leis. "ScaleStore: A fast and cost-efficient storage engine using DRAM, NVMe, and RDMA." In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 685–699.

[160] M. Liu, J. Kang, K. Wang, L. Zhang, H. Chen, X. Li, and T. Ding. "ScaleCache: Scalable and Production-Grade Buffer Management for Disk-Based Database Systems." *Proceedings of the VLDB Endowment*, **18**(12), 2025, pp. 5073–5085.

[161] X. Zhou, X. Yu, G. Graefe, and M. Stonebraker. "Two is better than one: The case for 2-tree for skewed data sets." *memory*, **11**, 2023, p. 13.

[162] X. Zhou, X. Hao, X. Yu, and M. Stonebraker. "Tiered-Indexing: Optimizing Access Methods for Skew: X. Zhou et al." *The VLDB Journal*, **34**(4), 2025, p. 45.

[163] J. J. Levandoski, D. B. Lomet, and S. Sengupta. "The Bw-Tree: A B-tree for new hardware platforms." In: *ICDE*. 2013, pp. 302–313.

[164] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. "Building a bw-tree takes more than just buzz words." In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 473–488.

[165] X. Hao and B. Chandramouli. "Bf-tree: A modern read-write-optimized concurrent larger-than-memory range index." *Proceedings of the VLDB Endowment*, **17**(11), 2024, pp. 3442–3455.

[166] K. Kim, T. Wang, R. Johnson, and I. Pandis. "Ermia: Fast memory-optimized database system for heterogeneous workloads." In: *SIGMOD*. 2016, pp. 1675–1687.

[167] F. M. Schuhknecht, J. Dittrich, and A. Sharma. "RUMA Has It: Rewired User-space Memory Access is Possible!" *Proceedings of the VLDB Endowment*, **9**(10), 2016, pp. 768–779.

[168] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. "Improving hash join performance through prefetching." *ACM Transactions on Database Systems (TODS)*, **32**(3), 2007, 17–es.

[169] J. Shim, J. Oh, H. Roh, J. Do, and S.-W. Lee. "Turbocharging Vector Databases using Modern SSDs." *Proceedings of the VLDB Endowment*, **18**(11), 2025, pp. 4710–4722.

[170] M. Wang, W. Xu, X. Yi, S. Wu, Z. Peng, X. Ke, Y. Gao, X. Xu, R. Guo, and C. Xie. "Starling: An i/o-efficient disk-resident graph index framework for high-dimensional vector similarity search on data segment." *Proceedings of the ACM on Management of Data*, **2**(1), 2024, pp. 1–27.

[171] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. "Can far memory improve job throughput?" In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[172] S. Jalalian, S. Patel, M. R. Hajidehi, M. Seltzer, and A. Fedorova. "{ExtMem}: Enabling {Application-Aware} Virtual Memory Management for {Data-Intensive} Applications." In: *2024 USENIX annual technical conference (USENIX ATC 24)*. 2024, pp. 397–408.

[173] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. "Efficient memory disaggregation with infiniswap." In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 649–667.

[174] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. "Semeru: A {Memory-Disaggregated} managed runtime." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 261–280.

[175] K. Zhong, W. Cui, X. Chen, Q. Li, Z. Yang, Y. Lu, X. Yan, S. Luo, Q. Yuan, and K. Huang. "Revisiting swapping in user-space with lightweight threading." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **42**(11), 2023, pp. 4205–4218.

[176] A. Papagiannis, M. Marazakis, and A. Bilas. "Memory-mapped I/O on steroids." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 277–293.

[177] I. Meignan, M. Misono, V. Leis, P. Bhatotia, et al. "{uCache}: A Customizable Unikernel-based {IO} Cache." In: *24th USENIX Conference on File and Storage Technologies (FAST 26)*. 2026, pp. 701–720.

[178] K. Zhao, S. Gong, and P. Fonseca. "On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 540–555.

[179] P. Pang, G. Deng, K. Bai, Q. Chen, S. Sun, B. Liu, Y. Xu, H. Yao, Z. Wang, X. Wang, et al. "Async-fork: Mitigating Query Latency Spikes Incurred by the Fork-based Snapshot Mechanism from the OS Level." *arXiv preprint arXiv:2301.05861*, 2023.

[180] M. Butrovich, W. S. Lim, L. Ma, J. Rollinson, W. Zhang, Y. Xia, and A. Pavlo. "Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems." In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 617–630.

[181] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. "Dune: Safe user-level access to privileged {CPU} features." In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 2012, pp. 335–348.

[182] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. "{OSv—Optimizing} the Operating System for Virtual Machines." In: *2014 usenix annual technical conference (usenix atc 14)*. 2014, pp. 61–72.

[183] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, et al. "Unikraft: fast, specialized unikernels the easy way." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 376–394.

[184] T. Mühlbauer, W. Rödiger, A. Kipf, A. Kemper, and T. Neumann. "High-performance main-memory database systems and modern virtualization: Friends or foes?" In: *Proceedings of the Fourth Workshop on Data analytics in the Cloud*. 2015, pp. 1–4.

[185] M. Grund, J. Schaffner, J. Krueger, J. Brunnert, and A. Zeier. "The effects of virtualization on main memory systems." In: *Proceedings of the sixth international workshop on data management on new hardware*. 2010, pp. 41–46.

[186] E. Deehr, W.-Q. Fang, H. Reza Taheri, and H.-F. Yun. "Performance analysis of database virtualization with the TPC-VMS benchmark." In: *Performance Characterization and Benchmarking. Traditional to Big Data: 6th TPC Technology Conference, TPCTC 2014, Hangzhou, China, September 1–5, 2014. Revised Selected Papers 6*. Springer. 2015, pp. 156–172.

[187] S. Bose, P. Mishra, P. Sethuraman, and R. Taheri. "Benchmarking database performance in a virtual environment." In: *Performance Evaluation and Benchmarking: First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers 1*. Springer. 2009, pp. 167–182.

[188] A. A. Soror, A. Aboulnaga, and K. Salem. "Database virtualization: A new frontier for database tuning and physical design." In: *2007 IEEE 23rd International Conference on Data Engineering Workshop*. IEEE. 2007, pp. 388–394.

[189] U. F. Minhas, J. Yadav, A. Aboulnaga, and K. Salem. "Database systems on virtual machines: How much do you lose?" In: *2008 IEEE 24th International Conference on Data Engineering Workshop*. IEEE. 2008, pp. 35–41.

[190] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. "The snowflake elastic data warehouse." In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 215–226.

[191] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, et al. "Amazon Redshift re-invented." In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 2205–2217.

[192] The gVisor Authors. *gVisor: The Container Security Platform*.

[193] G. Haas and V. Leis. "What Modern NVMe Storage Can Do, and How to Exploit It: High-Performance I/O for High-Performance Storage Engines." *PVLDB*, **16**(9), 2023, pp. 2090–2102.

[194] *io_uring 122M IOPS*. URL: https://www.reddit.com/r/linux/comments/wdlfjz/io_uring_122m_iops_in_2u_with_80_of_the_system/.

[195] *AWS Instance Types*. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html.

[196] *Azure BareMetal Infrastructure*. URL: https://learn.microsoft.com/en-us/azure/baremetal-infrastructure/.

[197] *Bare Metal Solution*. URL: https://cloud.google.com/bare-metal/docs.

[198] K. Fraser. *Practical lock-freedom*. Tech. rep. University of Cambridge, Computer Laboratory, 2004.

[199] T. A. Brown. "Reclaiming memory for lock-free data structures: There has to be a better way." In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. 2015, pp. 261–270.

[200] C.-E. Lin. *[RFC PATCH 0/6] Introduce Copy-On-Write to Page Table*. https://lore.kernel.org/lkml/20220521040301.GA1508050@strix-laptop/T/.

[201] C.-E. Lin. *[PATCH v4 00/14] Introduce Copy-On-Write to Page Table.* https://lore.kernel.org/lkml/CANOhDtU3J8SUCzKtKvPPPrUHyo+LV5npNObHtYP_AK4W3LomDw@mail.gmail.com/T/.

[202] *Micron Launches 9550 PCIe Gen5 SSDs: 14 GB/s with Massive Endurance.* URL: https://www.anandtech.com/show/21488/micron-launches-9550-pcie-gen5-ssds-14-gbs-with-massive-endurance.

[203] *Terabit Ethernet.* URL: https://en.wikipedia.org/wiki/Terabit_Ethernet.

[204] *Using Large Pages in KVM.* URL: https://www.linux-kvm.org/page/UsingLargePages.

[205] J. Dean and L. A. Barroso. "The tail at scale." *Communications of the ACM*, **56**(2), 2013, pp. 74–80.

[206] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. "Queues {don't} matter when you can {JUMP} them!" In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 1–14.

[207] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter. "{RobinHood}: Tail Latency Aware Caching–Dynamic Reallocation from {Cache-Rich} to {Cache-Poor}." In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 195–212.

[208] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. "Quickly generating billion-record synthetic databases." In: *SIGMOD*. 1994, pp. 243–252.

[209] A. Skiadopoulos, Q. Li, P. Kraft, K. Kaffes, D. Hong, S. Mathew, D. Bestor, M. Cafarella, V. Gadepally, G. Graefe, et al. "DBOS: a DBMS-oriented Operating System." 2021.

[210] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. "The Vertica Analytic Database: C-Store 7 Years Later." *PVLDB*, **5**(12), 2012.

[211] M. Stonebraker, S. Madden, and P. Dubey. "Intel "Big Data" Science and Technology Center Vision and Execution Plan." *SIGMOD Rec.*, **42**(1), May 2013, pp. 44–49.

[212] X. Zhou, X. Yu, G. Graefe, and M. Stonebraker. "Lotus: scalable multi-partition transactions on single-threaded partitioned databases." *PVLDB*, **15**(11), 2022, pp. 2939–2952.

[213] P. Kraft, Q. Li, K. Kaffes, A. Skiadopoulos, D. Kumar, D. Cho, J. Li, R. Redmond, N. Weckwerth, B. Xia, et al. "Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework." *arXiv preprint arXiv:2208.13068*, 2022.

[214] Q. Li, P. Kraft, M. Cafarella, Ç. Demiralp, G. Graefe, C. Kozyrakis, M. Stonebraker, L. Suresh, X. Yu, and M. Zaharia. "R3: Record-Replay-Retroaction for Database-Backed Applications." *PVLDB*, **16**(11), 2023, pp. 3085–3097.

[215] M. Müller and O. Spinczyk. "Mxkernel: rethinking operating system architecture for many-core hardware." In: *9th Workshop on Systems for Multi-core and Heterogenous Architectures*. 2019.

[216] J. Giceva, T.-I. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. "COD: Database/Operating System Co-Design." In: *CIDR*. 2013.

[217] J. Lin, T. Ji, X. Hao, H. Cha, Y. Le, X. Yu, and A. Akella. "Towards accelerating data intensive application's shuffle process using smartnics." *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, **7**(2), 2023, pp. 1–23.

[218] Q. Zhang, P. Bernstein, B. Chandramouli, J. Hu, and Y. Zheng. "Dds: Dpu-optimized disaggregated storage." *arXiv preprint arXiv:2407.13618*, 2024.

[219] Y. Zhong, H. Li, Y. J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, et al. "{XRP}:{In-Kernel} Storage Functions with {eBPF}." In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 375–393.

[220] F. Schuhknecht and J. Henneberg. "Towards Adaptive Storage Views in Virtual Memory." *arXiv preprint arXiv:2209.01635*, 2022.

[221] F. Schuhknecht and J. Henneberg. "Accelerating Main-Memory Table Scans with Partial Virtual Views." In: *Proceedings of the 19th International Workshop on Data Management on New Hardware*. 2023, pp. 89–93.

[222] D. De Leo and P. Boncz. "Packed memory arrays-rewired." In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE. 2019, pp. 830–841.

[223] F. Schuhknecht. "Taking the Shortcut: Actively Incorporating the Virtual Memory Index of the OS to Hardware-Accelerate Database Indexing." 2024.

[224] E. Bendersky. *Measuring context switching and memory overheads for Linux threads*. 2018.