Practical DB-OS Co-Design with Privileged Kernel Bypass

Xinjing Zhou
MIT CSAIL
xinjing@mit.edu

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

Jinming Hu
DolphinDB
conanhujinming@gmail.com

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

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

Abstract

This paper revisits the longstanding challenge of coordinating database systems with general-purpose OS interfaces, such as POSIX, which often lack tailored support for DB requirements. Existing approaches to this DB-OS co-design struggle with limited design space, security risks, and compatibility issues. To overcome these hurdles, we propose a new co-design approach leveraging virtualization to elevate the privilege level of DB processes. Our method enables database systems to fully exploit hardware capabilities via virtualization, while minimizing the need for extensive modifications to the host OS kernel, thereby maintaining compatibility. We demonstrate the effectiveness of our approach through two novel virtual memory mechanisms tailored for database workloads: (1) an efficient snapshotting mechanism that captures memory snapshots at millisecond intervals for in-memory databases and HTAP workloads, and (2) a streamlined in-kernel buffer pool design. We introduce LibDBOS, a lightweight guest kernel implementing these mechanisms. Our evaluations highlight significant improvements in latency and efficiency compared to existing snapshotting and buffer pool designs, underscoring the potential of the approach.

1 INTRODUCTION

Database systems often aim to tightly control hardware to optimize performance, as they typically have better insights into the workloads than the operating system. However, with the performance gap between CPUs and modern I/O devices narrowing, database systems are increasingly dissatisfied with the general-purpose interfaces provided by the OS. For instance, virtual memory snapshotting has been employed in systems like Hyper [44] and Redis. Hyper originally utilized `fork` to isolate OLAP queries from OLTP by running OLAP queries on a forked OLTP process. Nevertheless, Hyper eventually transitioned to software-based MVCC due to the high overheads and lack of control over the semantics of `fork`. Recent research [25] also highlighted that `mmap` (a POSIX API) and the Linux page cache are unsuitable for building buffer pools for data caching because of their limited control over page eviction and inherent performance issues when operating on fast storage devices. As a result, modern high-performance OLTP systems often bypass `mmap` in favor of software-based page caching, forfeiting the advantages of hardware-assisted translation. In fact, interface mismatches of this kind have been recognized for decades [36, 67].

Broadly speaking, there are two main approaches to co-designing a database system with an OS. The first approach involves designing and maintaining custom OS kernels specifically tailored for databases [71, 72]. However, this approach is generally unsustainable due to the high maintenance overhead associated with supporting drivers for newly released hardware. Recent work [50] explores the use of cloud-focused Unikernels – single-address-space kernels without user-kernel privilege isolation—for co-designing OS and database systems in cloud environments. The second approach is to extend an existing general-purpose OS, such as Linux, with specialized interfaces for database systems. This approach is more sustainable as the OS abstracts devices and microarchitectures. Currently, most co-designs focus on extending the Linux kernel by adding new system calls/modules [58, 64, 75], specializing kernel subsystems [49], or incorporating user logic into the kernel via modern Linux eBPF infrastructure [22–24, 34, 79]. Nevertheless, these approaches still face at least one of three major limitations.

Problem #1: Limited Design Space. Traditional OS kernels such as Linux maintain privilege separation between the DBMS, which runs in user space, and the OS, which runs in kernel space. Hence, the DBMS must still cross the costly privilege boundary [80]. Moreover, eBPF only allows verified user code to run in the kernel (e.g., Unbounded loop and floating-point instructions are not allowed) with restricted programming model and APIs [5, 6, 13].

Problem #2: Security and Maintainability Concerns. Making ad-hoc changes to complex kernels is hard, potentially enlarging the attack surface and introducing catastrophic security vulnerabilities [1]. Such ad-hoc changes will also increase the maintenance burden, slow down the evolution of the kernel itself, and complicate the integration of upstream Linux kernel updates.

Problem #3: Compatibility and Ecosystem Issues. While the Unikernel-based approach can increase the co-design space by running DBMS in the single address space with the kernel, the architecture implies that well-known tooling (e.g., monitoring and debugging [15, 32]) and rich ecosystem around a mature multi-process OS (e.g., Linux) is no longer available.

Is it possible to fully harness the power of hardware for DBMS while minimizing security and maintainability issues without abandoning the Linux ecosystem? This paper proposes a co-design paradigm using a privileged DB process to address these limitations. In this paradigm, the DBMS runs in the kernel space of a lightweight guest OS on top of a hypervisor, addressing the first two challenges. First, the DBMS in kernel space has direct access to virtualized hardware resources (e.g., privilege levels, virtual memory, IOMMU, and interrupts) without system call overhead, enabling abstractions that are infeasible in user space. Second, changes introduced by these abstractions only affect the guest kernel, ensuring the stability of the host kernel. To address the third issue, we utilize a specific

hypervisor [17] that exports hardware-assisted virtualization (e.g., Intel VT-x) to a Linux process, granting the process privileged status. This hypervisor, implemented as a Linux kernel module, proxies system calls from the guest process to the host kernel, preserving process abstraction and POSIX compatibility for DBMS. We refer to this paradigm as **co-design with privileged kernel bypass**.

To demonstrate the potential of our approach, we present two novel virtual memory mechanisms for database systems enabled by this co-design. First, we identify the bottleneck in the Linux `fork` system call and introduce a snapshotting mechanism, Snappy, which can capture memory snapshots almost instantaneously at millisecond-level frequencies—something impractical in the Linux virtual memory subsystem due to its complexity and design constraints. In our co-design paradigm, this becomes feasible by redesigning the virtual memory subsystem within the DB process, specifically tailored for the DBMS, while still leveraging components of the host kernel. When Snappy was evaluated on Redis for checkpointing, we observed orders-of-magnitude improvements in tail query latency during ongoing checkpoint processes. Second, we propose Tabby, an in-kernel buffer pool design that tightly integrates virtual memory hardware for page translation, maintaining control over eviction without incurring the overhead of TLB shootdown or unscalable memory allocation [25]. Tabby delivers state-of-the-art performance and efficiency in both in-memory and out-of-memory workloads. To facilitate the execution of existing database systems in a virtualized environment and to leverage these new mechanisms, we developed a minimal guest kernel called Libdbos that implements these innovations.

This paper makes several key contributions. First, we are the first to present such a co-design paradigm with privileged DB process, which allows database system designers to unlock new possibilities without compromising the security or maintainability of the host OS kernel or abandoning the host OS ecosystem. Second, we present two unique DB-OS mechanisms enabled by this co-design: a kernel-based buffer pool design and an instantaneous, high-frequency snapshotting mechanism. Third, we develop Libdbos, a guest kernel that implements these new mechanisms that can be used by existing database systems. Finally, we conduct extensive evaluations to demonstrate the effectiveness of the proposed abstractions.

2 BACKGROUND AND MOTIVATION

This section covers the necessary motivation for co-design and the background of virtualization. Specifically, we use virtual memory to motivate the need for better co-design and explain why existing co-design approaches are insufficient.

2.1 DB-OS Interface Mismatch

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

Virtual Memory Snapshot. Memory snapshotting, such as the Unix `fork`, is a valuable OS abstraction used in various database applications. For instance, Redis employs `fork` to snapshot its in-memory database, which is then serialized to storage as checkpoints. Similarly, the HTAP database system HyPer [44] uses `fork` to run OLAP queries on a snapshot of its in-memory OLTP database, enabling workload isolation. These systems rely on the efficient

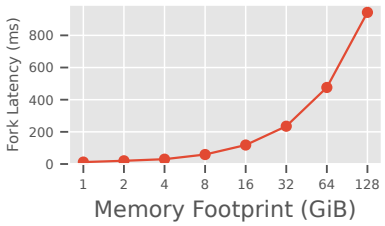


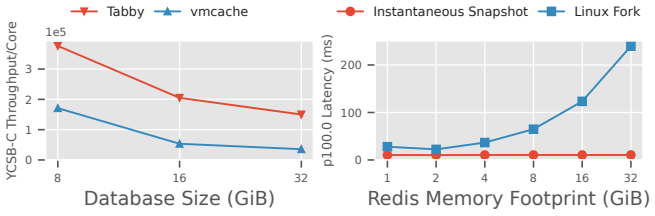
Figure 1: Fork Latency varying Memory Footprint

```
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 2: CPU Cycle Distribution in `copy_pte_range` in Linux Kernel: most of the cycles are spent in memory accesses to update reference counts

implementation of `fork`, typically achieved through the Copy-on-Write (CoW) technique on modern OSes. However, in Linux, `fork` is synchronous and single-threaded, with latency that grows linearly with the memory footprint of the forked process, as shown in Figure 1. Even for a moderate memory footprint (tens of GBs), latency can be significantly high. To obtain a consistent memory snapshot, the application must pause latency-sensitive threads involved in query and transaction processing, such as those in Redis and HyPer, leading to substantial stalls and increased tail latency. Figure 3b illustrates how the tail latency of Redis write queries rises sharply during checkpointing with `fork`. Additionally, the high fork latency forces systems to take snapshots at intervals of seconds (HyPer) to minutes (Redis). While this may suffice for checkpointing in Redis, it is problematic for HTAP databases like HyPer, as the low snapshot frequency means some OLAP queries are executed on stale data. A low-latency, high-frequency snapshot mechanism would allow a VM-snapshot-based HTAP application to serve queries on much fresher data.

Intuitively, the majority of time in a CoW-based `fork` implementation should be spent copying the page table. With a 64 GB memory footprint, the page table size is approximately 128 MB, assuming 4 KB pages and 8-byte last-level page table entries (PTEs). Given our machine's measured sequential memory bandwidth of 8 GB/s with a single thread, this task should take only 16 milliseconds. However, as shown in Figure 1, it actually took about 500 milliseconds. To investigate this discrepancy, we used `perf` to measure the cycle distribution of the benchmark program during the `fork` system call. Surprisingly, 98% of the cycles were spent in the `copy_pte_range` kernel function. Figure 2 highlights the hot spots in the assembly code. Upon analyzing the source code, we discovered that these hot spots correspond to memory accesses used to increment the reference counts of physical pages pointed to by the PTEs. While copying the page table primarily involves sequential memory access, the metadata for physical pages is organized by physical memory addresses rather than virtual ones, resulting in random memory access. These reference counts, essential for managing the lifecycle of physical pages, are critical for supporting



(a) Buffer management on fast SSDs (64 Threads, 8GB Buffer Pool) **(b) Tail latency of write queries on Redis during checkpoint**

Figure 3: Benefits of DB-OS Co-design.

key Linux VM features such as shared memory, the page cache, swapping, and memory-mapped files. In contrast, our proposed snapshot mechanism, which avoids reference counting, drastically reduces snapshot latency in Redis, as shown in Figure 3.

MMAP-based Buffer Management. Some database systems use `mmap`-based APIs for file system I/O and buffer management, exploiting virtual memory's ability to address more memory than the physical capacity. This approach relies on the OS (e.g., Linux kernel) to cache database file pages. Compared to a traditional buffer pool, which manages a set of hot pages identified by logical page IDs, the `mmap`-based approach eliminates a layer of indirection (e.g., software hash tables) by utilizing virtual memory hardware, improving performance when the working set fits in memory and the workload is read-only. However, as noted in recent research [25], `mmap` VM APIs exposed by traditional OSes suffer from significant issues when data exceeds memory capacity or when workloads are not read-only. These issues include performance bottlenecks from TLB shootdowns and page allocation, as well as a lack of APIs for ARIES-style transactional safety. vmcache [49] demonstrated that page eviction and I/O can be controlled in user space while still leveraging virtual memory hardware for translation through clever use of `mmap` APIs. However, without modifications to the kernel memory subsystem, vmcache remains constrained by the Linux memory allocator and TLB shootdowns during high-speed virtual memory operations. As shown in Figure 3a, on an out-of-memory YCSB-C workload, vmcache's throughput per core is significantly lower than our co-designed buffer pool, Tabby, as vmcache spends 70% of its cycles on TLB shootdowns in the Linux kernel.

2.2 Why are Alternatives not Sufficient?

Next, we discuss existing solutions and explain why they do not sufficiently address the challenges raised in the introduction, and summarize their strength and weaknesses in Table 1.

Specializing Linux Kernel. Most co-design follows this approach. For instance, the `exmap` module in vmcache [49] specializes the Linux memory subsystem to enable efficient allocation and batch TLB shootdowns. Anker [64] introduced a system call to bypass the limitations of `mmap` for fine-grained memory snapshots. However, fundamental design principles of the Linux kernel restrict the extent of possible modifications. For example, eliminating page reference counting is practically impossible without a major redesign of the Linux virtual memory subsystem. Thus, the co-design space is constrained by the legacy code and underlying assumptions of Linux. Moreover, the complexity of the Linux virtual memory subsystem demands extensive familiarity with the kernel to ensure that any modifications are both safe and functional.

Kernel Bypass. Another common approach to reducing OS overhead is bypassing the OS kernel in database systems, by implementing performance-critical components (e.g., file systems and network stacks) in user space and utilizing I/O libraries such as DPDK or SPDK. However, since these I/O libraries run in user space, they cannot directly program interrupt hardware. As a result, they rely on spin-polling to complete requests, which is inefficient under low load. Additionally, user space cannot directly control virtual memory hardware, requiring the use of software-based translation tables for buffer pools or complex pointer-swizzling techniques. This leads to suboptimal performance and increased complexity for the database system. In summary, kernel bypass leaves many performance and simplicity opportunities unexploited.

User Bypass. Recent work [24] has also explored moving frequently used, data-intensive components (e.g., B-tree traversal and DB proxy) into kernel space by leveraging the eBPF infrastructure. A key research challenge is that eBPF only supports user programs that are verifiably safe. For instance, loops must be bounded, and programs are restricted to using a limited set of kernel data structures for stateful operations. Additionally, the Linux kernel prohibits the use of floating-point operations, which are essential for many database applications. As a result, existing database systems must be rewritten using a constrained programming model that functions within the Linux kernel.

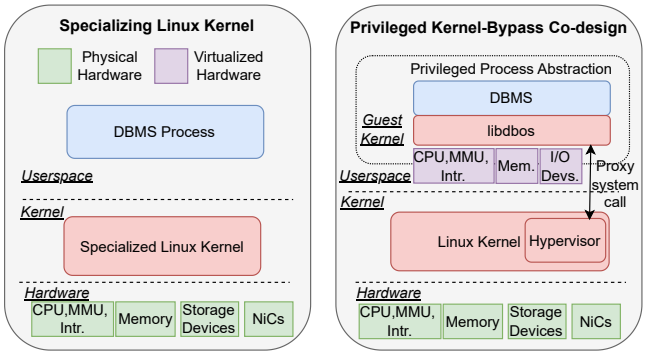
Co-design with Unikernel. Recent work [50] envisions running DBMS on a Unikernel atop a virtual machine (VM) in the cloud. A Unikernel [45, 47, 54] is a minimal OS kernel designed to run a single application in a single address space and privilege level. This minimalistic design is suited for deployment on a VM, where the hypervisor manages isolation and deployment. A key consequence of this architecture is that the application gains direct access to hardware and privileged instructions. In this setup, the database operates at the same privilege level as the kernel, and system calls are treated as ordinary functions handled directly by the guest kernel. However, Unikernel requires the implementation of drivers for virtualized hardware and POSIX compatibility for databases that depend on POSIX APIs. Moreover, the tooling and ecosystem surrounding Unikernel are still underdeveloped.

2.3 Virtualization

Hardware Assisted Virtualization. Although the mismatch was identified decades ago and still persists, virtualization has become central to modern data centers and cloud vendors, transforming resource management and deployment. By dividing a physical machine into multiple virtual machines, virtualization improves resource utilization, security isolation, and ease of deployment. Initially, virtualization was seen as having high performance overhead compared to bare-metal. However, hardware-assisted virtualization, like Intel VT-x and AMD-v, has largely closed this gap, allowing virtual CPUs to run at native speed. Extended Page Tables (EPT) enhance memory management by reducing costly hypervisor round-trips for page faults. IO-virtualization (e.g., SR-IOV) enables devices like network adapters or SSDs to bypass the host OS's I/O stack, improving performance. Studies [20, 30, 39, 55, 56, 66] show that virtualized environments impose minimal overhead on both OLTP and OLAP workloads. Most cloud database offerings [16, 27, 70]

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

Table 1: Qualitative Comparison of DB-OS Co-design Approaches.



(a) Specializing Linux Kernel **(b) Privileged Kernel-Bypass**

Figure 4: Comparison of Co-design Architectures.

are now deployed in virtualized environments, demonstrating that virtualization introduces little performance impact.

Privileged Process via Virtualization. Another novel way of leveraging virtualization is to make a Linux process privileged, pioneered by Dune [17], 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 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.

3 PRIVILEGED KERNEL-BYPASS CO-DESIGN

In this section, we describe and analyze high-level paradigms for DB-OS co-design and show how the proposed paradigm differs. Figure 4 contrasts the traditional co-design approach of specializing the Linux kernel with the proposed privileged DB process approach. In traditional co-design (Figure 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 1 shows, specializing the kernel directly has significant security and maintainability issues, which may explain why this approach has not gained widespread adoption.

Figure 4b illustrates privileged kernel-bypass 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 [17, 69], 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 [17] 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), and 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 proxy 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 4) and Tabby (Section 5). 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.

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 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 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 page cache, shared memory, and memory-mapped files.

Taking Advantages of Linux Kernel Advances. Previous research has also 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 [9, 40]. 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 the privileged DB process, 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. Our kernel buffer pool, Tabby (Section 5), reuses the Linux I/O path while bypassing the slow virtual memory manipulation.

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 privileged DB process. **On premise**, users have control over physical hardware and can install the hypervisor for privileged DB process. Another option is to leverage **Bare-metal cloud instances**, which are widely available in all major cloud vendors [2–4] where users can install the 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.

3.3 Comparison to Unikernel-based Co-design

Compared to Unikernel-based co-design, they have the same co-design space as both run the DBMS in kernel space, and they have lower security guarantees than bypass mechanisms as DBMS runs in the kernel space (though virtualized). 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. The proposed approach, 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 1.

3.4 More Opportunities

While the paper presents only two examples of new abstractions, there is much more potential than meets the eye. We next describe more opportunities enabled by our co-design paradigm:

Fast Memory-Rewiring for Query Processing and Indexing. Memory-rewiring [63], a technique that remaps existing virtual memory mappings in user space, has been applied to query processing [61, 62] and indexing [28, 60]. However, memory-rewiring heavily relies on `mmap` to create mappings in Linux, which carries a fundamental cost of updating the reference count of each physical page in the mapped region, as we showed in the analysis of Section 2.1. As we will show later in Section 4, which described a snapshot-ting mechanism, its core technique can be used for building a faster memory-rewiring as it does not use a per-page reference count in its specialized virtual memory system. Hence, our co-design has the potential to make this technique more practical.

Synergy with Existing Bypass Mechanisms There have been many recent efforts [22–24, 34, 79] in the research community to leverage bypass mechanisms around the networking and storage stacks to accelerate data-intensive applications. DBMS designers can consider combining privileged process and existing bypass efforts because privileged DB process can still issue system calls, which are the interfaces for interacting with these bypass mechanisms. Privileged process allows easy specialization of CPU and virtual memory subsystems, which is hard to do with bypass mechanisms due to security concerns of the Linux kernel. For example, on top of a DBMS running in a privileged process with a specialized virtual memory subsystem, one may leverage eBPF-based caching solutions [79] for simple reads on hot records, reducing the networking overhead.

Security Isolation for User-defined Logic in DBMS. Many DBMSs support extending functionality through user-defined logic (e.g., stored procedure, UDF, loadable modules/extensions), which typically run in the same address space (user-space) as the DBMS for performance and interoperability. This introduces security challenges for shared DBMS (e.g., multi-tenant cloud DB) that must guard against malicious user code for accessing confidential information. With DBMS running as a privileged process, one can leverage virtual memory and privilege levels to confine the execution of user-defined logic in user space while placing DBMS in the kernel space, achieving memory isolation.

Lightweight Preemptive Threading. The DBMS running in a privileged DB process can leverage interrupt hardware and system call interception to build a lightweight preemptive scheduler with much lower overhead compared to the expensive scheduling of Linux kernel threads [18]. This is not possible in user-space threading libraries, as user-space cannot directly program the interrupt hardware for preemption. This helps with increasing the concurrency of the DBMS to saturate high-end I/O devices [40].

4 Snappy: 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. As the analysis in Figure 2 has shown, 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 VM system. 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 snapshot-ting. Next, we show a lightweight scheme of Snappy for creating, managing, and destroying snapshots at high frequency without using reference count.

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. Only until the last reference is dropped is it safe to return the physical page to the system allocator. With the absence of 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,

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

we show how to efficiently track the references in a batch fashion, inspired by epoch-based reclamation [21, 33].

The pseudo-code of our algorithms is listed in Figure 5. We associate each snapshot page table with an epoch number allocated from a monotonically increasing global timestamp when the snapshot table is copied from the main page table used by the OLTP workers and loaded by a CPU core in `dbos_snappy_spawn`. This epoch number establishes an invariant that any PTE copied before the epoch number is assigned might be referenced. 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 might have 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.

4.2 Instantaneous Snapshot

While snapshot creation is significantly faster now as the random memory access has been drastically reduced, the copying 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. 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 CPU without doing further processing. When a snapshot is requested, the user may retrieve one ready snapshot 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.

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 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 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)

# Hanlders registered at program start
def dbos_snappy_page_fault_handler(pgtbl, vaddr): # Section 4.1
    if write-protection fault and pgtbl is from snapshots:
        newp = copy page at vaddr
        update pgtbl at vaddr to point to new p
    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 4.4
    tlb-shootdown on vaddr and synchronize main_pgtl changes to
    ↪ inactive snapshots # Section 4.2/4.4

def dbos_snappy_syscall_handler(tf : Trapframe): # Section 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: Algorithms for Epoch-based Snapshot Management

4.3 Challenge: Supporting Multiple Page Tables

One unexpected challenge we faced when implementing such a VM system in a privileged process is to support multiple page tables, which is not covered in the original paper [17]. One of the fundamental design decisions in a privileged process is to preserve the ability to issue system calls to the host operating system. Dune hypervisor needs to maintain the invariant that the same virtual memory address mapping to the host physical memory with the mapping represented by the kernel page table. As illustrated in Figure 6, Dune ensures that guest virtual address (GVA) 0x8000 from Page Table 1 inside the privileged process maps to the same host physical address (GPA) 0x3000 to which the kernel page table for the privileged process maps the same host virtual address (HVA) 0x8000. Note that the translation from GPA to HPA is accelerated with an extended page table (EPT) by the hypervisor on most hardware. This ensures that system calls that read/write from/to userspace memory work without modifications to the application.

However, this invariant might be violated when introducing another page table. For example, shown in Figure 6, assume that the core working on the snapshot is loaded with Page Table 2 and triggers a CoW fault at address GVA 0x8000. The fault handler allocates a new page at GPA 0x5000 and installs a new mapping in Page Table 2 from GVA 0x8000 to GPA 0x5000. Later, access to GPA 0x5000 triggers an EPT page fault, creating a host physical page at HVA 0x2000 and mapping from GPA 0x5000 to HPA 0x2000. When the core wants to write the content at GVA 0x8000 to storage, it makes system call `write`, which takes in a user-space buffer. Since programs only deal with virtual addresses, it passes GVA 0x8000

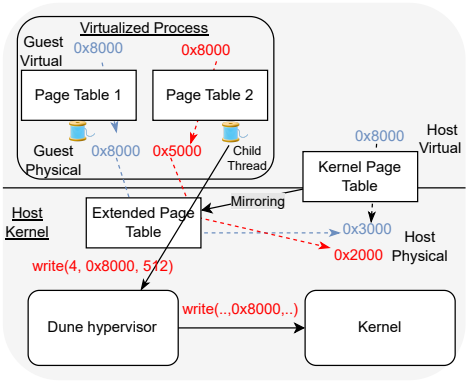


Figure 6: Problems of Dune with Multiple Page Tables: When the child core makes a write system call that passes a guest virtual address that intends to use mapping from Page Table 2 and EPT (GVA 0x8000→ HVA 0x5000→ HPA 0x2000), the kernel might erroneously use the translation of (HVA 0x8000→ HPA 0x3000).

as the `buf` argument. The kernel then erroneously uses mapping (0x8000→0x3000) from the kernel page table instead of using HPA 0x2000 for performing the I/O operation, resulting in inconsistency. The problem is that the same GVA might point to different GPA depending on the user page table used.

To address this problem without modifying the application, LibDBOS intercepts every system call that might access user-space memory made by the application. LibDBOS internally keeps a staging buffer **B** that bypasses the CoW mechanism. At intercepting such system call, LibDBOS copies data pointed by the user-space pointer to **B** and passes the address in the **B** down to the hypervisor for performing real system call. Since **B** will not be copy-on-writed, it's virtual to physical mapping is the same across all user page tables. Similarly, LibDBOS copies data back to the user space buffer when the host kernel writes to user-space memory (e.g., `read` system call). This is a general solution that guarantees application correctness with respect to system calls transparently at the cost of an extra copy. We discuss possible optimizations in the next section.

4.4 Prioritized Copy-on-Write

In this section, we show how to avoid the extra copy for system calls due to supporting multiple page tables in a privileged process. We can completely avoid copying to the staging buffer for latency-sensitive threads. The idea is that when a core with a main page table triggers a write-protection fault, instead of changing the PTE of the main table to point to a new physical page, we change the PTE of the snapshots. A direct implication of this is that the TLB shootdown involving all the cores with the main page table is eliminated. Instead, the faulting core sends a less costly TLB shootdown to the core working on the snapshots. We call such an approach Prioritized-Copy-on-Write (PCoW), and it favors the cores with the main page table. With PCoW, we can safely eliminate the extra copy to/from the staging buffer when system calls come from such cores. This is because these cores, upon a write-protection fault (shown in `dbos_snappy_page_fault_handler` function of Figure 5), do not change their faulting PTE to point to a new guest physical page. Therefore, its mapping always matches the host kernel page table. Such prioritization is beneficial because cores with the main page table are more latency-sensitive than child cores. For example, in Redis and Hyper, query and transaction processing happen on the main page table, directly impacting request latency.

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 [75], applied copy-on-write (CoW) to the page table itself. However, this proposal faced resistance from kernel maintainers [52, 53] due to concerns about potential instability and incompatibility. Another recent approach, async-fork [58], asynchronously copies page table 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.

5 Tabby: Kernel Buffer Pool without TLB Shootdown

In this section, we shift gear to talk about TLB shootdown, the general mechanism required in 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 shootdown or high memory allocation overhead. This is uniquely enabled by our co-design paradigm, which allows DB to access privileged instructions (TLB invalidation).

As Figure 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.

5.1 Eliminating TLB-Shootdowns

In Linux, a TLB-shootdown is needed whenever a PTE is modified. For example, vmcache buffer manager design [49] uses `madvise` system call to remove a physical page from the page table, triggering an unscalable shootdown for every page eviction. This raises the

[3]Measured in the `mm` directory of Linux kernel v5.15.90
[4]Measured by counting the `get_page`/`put_page` invocations in the code base

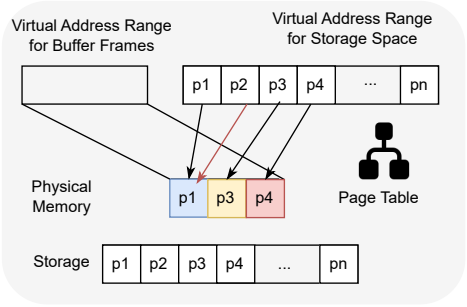


Figure 7: Tabby Buffer Pool: Tabby allows multiple virtual pages to be mapped to a physical frame. Incorrect mappings are fixed lazily.

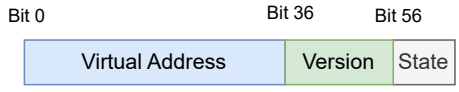


Figure 8: Tabby Page Header Format

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 9: Algorithms for Tabby Buffer Pool

question of whether it would be possible to leave the PTE (e.g., Page 2 in Figure 7) unchanged on eviction. A direct implication of this is that TLB shootdown is not needed. However, later access to the virtual page might erroneously result in accessing the wrong physical page using the stale mapping. 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,

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 intialization
def dbos_tabby_page_fault_handler(addr):
    pte = pgtbl_lookup(addr)
    if pte->lock() == False: # One bit of PTE used for lokcing
        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 10: Page Fault Handling in Tabby

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 8. We next show how to use this small metadata 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 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 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 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 (Lines 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). Writing the page with the lock bit on is to maintain 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 [49], Tabby supports optimistically reading a page without writing to shared memory for better scalability. As shown in `dbos_tabby_optimistc_read` in Figure 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 (Lines 28-29). Compared to vmcache, Tabby needs to additionally verify that the address did not change during the read (31-32).

5.2 Physical Memory Allocation

Linux kernel, designed to support multi-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.

5.3 Discussion

Tabby supports all the functionalities vmcache provides without the overhead of TLB shootdown and memory allocator. 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. [49] proposed a Linux kernel module called exmap that specializes the virtual memory subsystem in Linux kernel to address the bottlenecks by batching TLB shootdowns. While exmap reduces the number of TLB shootdowns, each shootdown still requires every CPU core to flush its local TLB, negatively impacting the TLB efficiency of modern CPUs (see Section 7.4). With modern storage [9, 10], network devices [12], and tiered memory [41, 59, 78] capable of hundreds of millions of IOPS, the costs of TLB shootdowns remain significant, even with batching. In contrast, Tabby preserves TLB efficiency by eliminating shootdowns. Beyond performance improvements, Tabby also has a smaller impact on host kernel security compared to exmap.

6 LibDBOS IMPLEMENTATION

6.1 LibDBOS Guest Kernel

We leveraged the open-source `libdune` [17] to implement this co-design paradigm as LibDBOS. We made about 3000 lines of change. Besides basic guest OS utilities such as page table manipulation, exception handling, 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 IPIs between vCPUs in the guest using posted interrupt, an Intel VT-d feature that allows a virtual CPU to receive interrupt without hypervisor intervention. Based on this, we implemented a basic TLB shootdown mechanism using IPI. When a vCPU intends to ask another vCPU to flush remote TLBs, the vCPU registers a function call to be executed on the remote vCPU and then sends an IPI to the target vCPU. 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. We leverage this entry point to perform custom system-call interposition.

6.2 LibDBOS Hypervisor

We built the hypervisor for LibDBOS on top of Dune [17] for the x86 architecture and made about 1250 lines of change. We added a few enhancements to the original hypervisor:

Host Huge Pages. To reduce the overhead of two-dimensional TLB miss overhead in a virtualized environment, we leverage huge pages of the host to back the physical memory of the process. This is common practice in modern hypervisor implementation [14].

Guest Huge Pages. Similarly, there is another layer of address translation inside the guest, which adds TLB-miss overhead. LibDBOS provides huge page support to reduce this layer of TLB miss overhead. It intercepts `mmap` calls via its system call interposition capability and allocates huge guest pages for mappings larger than 16 MB. This reduces the page walk length on the guest page table.

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 kernel with Snappy's custom page fault handling, as described in Figure 5. A background copy worker is created to maintain a set of (2 by default) ready snapshot page tables in the guest kernel. When a snapshot is requested, the background worker starts copying the page table from the main page table. To maintain synchronization during page table copying and updates, we utilize table-level locks. To handle system calls correctly (Section 4.3), we override the default system call handler to differentiate between system calls made by CPU cores using the main page table and those made on snapshots. For the latter, before proxy-ing the system calls to the host kernel, we copy the user buffers in the parameters of the system calls to and from a per-CPU staging buffer.

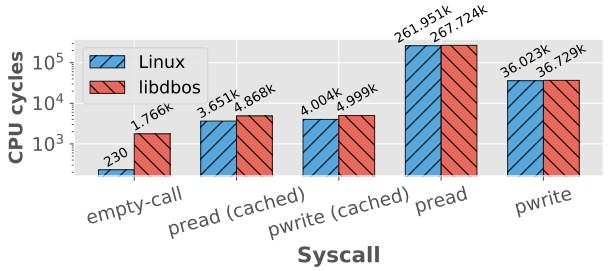


Figure 11: Impact of Virtualization on System Call

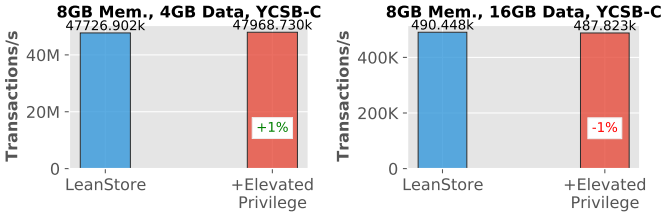


Figure 12: Impact of Privilege Elevation on LeanStore

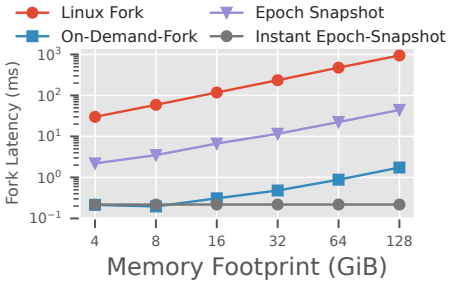
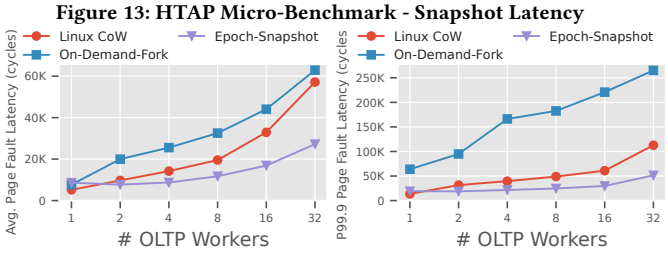


Figure 13: HTAP Micro-Benchmark - Snapshot Latency



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

Figure 14: HTAP Micro-Benchmark - OLTP Latency after Snapshot

6.4 Tabby Implementation

We developed Tabby by adapting the vmcache code base[5]. When Tabby is initialized, we replace the default page fault handler with Tabby's custom page fault handling, as described in Figure 10. We made about approximately 1,300 lines of modifications. We implemented a Clock replacement for 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.

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 Snappy 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 compared to the existing state-of-the-art buffer pool designs?

7.1 Baselines and Experimental Setup

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 for `BGSAVE` [6] command. We then compare it against running Redis on an unmodified Linux kernel (v6.6.1) and a modified Linux kernel (v6.0.6) [7] which implements On-Demand-Fork [75]. 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 [58] as we could not obtain the source code.

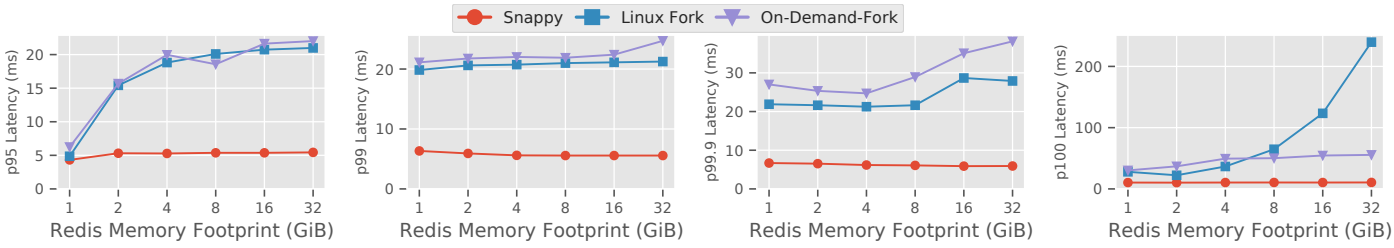
Buffer Management. We compare Tabby with the following systems: 1) ***vmcache***, a state-of-the-art buffer management approach that leverages virtual memory hardware for translation using POSIX `mmap` APIs. For evaluation, we use vmcache at commit `a8ed2b0`. 2) ***LeanStore***, a buffer pool design that utilizes pointer-swizzling to eliminate the hash table lookup overhead for memory-resident pages. We evaluate LeanStore at commit `dd42514`, configured with the Read Uncommitted isolation level. 3) ***WiredTiger***, a traditional buffer pool design that uses a software hash table for page table lookups. We evaluate WiredTiger v2.6.1, configured with the lowest isolation level, Snapshot Isolation.

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. 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.

7.2 Impact of Privilege Elevation

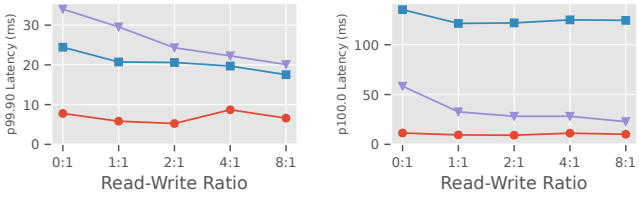
We begin by analyzing the performance impact of virtualization and privilege elevation on system calls, with a focus on I/O operations such as `pread` and `pwrite` that are used by most DBMSes. For comparison, we also measured the latency of an empty system call (`gettid`), representing the worst-case scenario for a privileged DB process. The results, as shown in Figure 11, indicate that virtualization imposes a fixed overhead of 1,536 CPU cycles, mainly used in exiting and entering the virtualization boundary for proxying system calls. For I/O system calls, the performance impact is much less. For example for un-cached `pread` and `pwrite` system

[5] https://github.com/viktorleis/vmcache
[6] https://redis.io/docs/latest/commands/bgsave/
[7] https://github.com/rssys/on-demand-fork



(a) P95 Query Latency (b) P99 Query Latency (c) P99.9 Query Latency (d) Maximum Query Latency

Figure 15: Tail Latency of 3M Write Queries varying Database Size (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 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.

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 LenaStore on Linux with YCSB-C workloads and 64 threads. The results are shown in Figure 12. When data fits in memory, we did not observe a measurable throughput difference. When the buffer pool can not cache all data, privileged LeanStore performs only 1% worse than LeanStore running in the user space of Linux due to VM exit/entry. Hence, without judiciously leveraging the hardware constructs available, running at a higher privilege level does not translate to higher performance.

7.3 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 CoW (PCoW) discussed in Section 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 20a, 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.

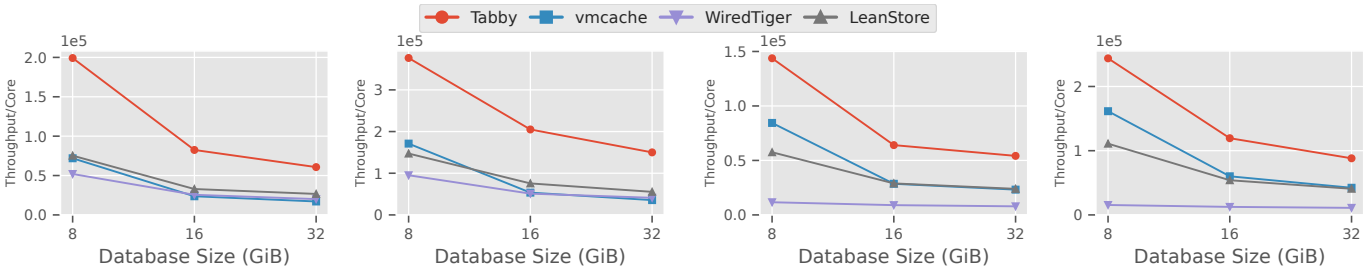
OLTP Latency. To understand the impact of snapshot and CoW on OLTP threads, we measure the latency to resolve a CoW fault page fault. We show the results in Figure 14a and Figure 14b. Generally, `Epoch-Snapshot` generally outperforms Linux and On-demand-fork as the number of OLTP threads increases by 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 need to send TLB-shootdown when a process has only one thread. Whereas, 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 Benchmark. Next, we evaluate the impact of snapshot mechanisms when applied to a real-world in-memory database system, Redis. For performance metric, we focus on tail latency, including maximum latency, as they are important metrics for in-memory caching/database for user-facing services [19, 29, 38]. We enable all optimizations for Snappy.

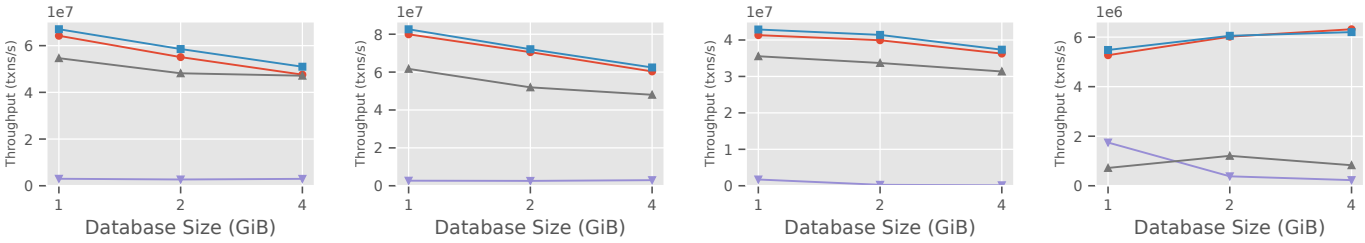
Uniform Workload. We first run an experiment studying the write query latency after a snapshot is taken while varying the database size. We use `memtier_benchmark`[8] to generate workload traffic. We issue a total of 3×10^6 queries to a pre-populated and pre-warmed Redis server while keeping a query 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 15. We can see that Snappy, compared to unmodified Linux, significantly reduces latency across all the measured percentile points by as much as 15× on maximum latency, achieving

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



(a) YCSB-C Uniform (b) YCSB-C Skew (c) YCSB-A Uniform (d) YCSB-A Skew

Figure 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.



(a) YCSB-C Uniform (b) YCSB-C Skew (c) YCSB-A Uniform (d) YCSB-A Skew

Figure 18: Throughput of different buffer pool designs on in-memory workloads. (8GB Buffer Pool, 64 Threads)

the lowest tail latency. Surprisingly, On-Demand-Fork is only effective for the maximum latency (Figure 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. Snappy, on the other hand, performs page-table copying work in advance and does not impose additional processing overhead on normal CoW faults.

Skewed Workload. Real-world workloads exhibit skewed patterns. Therefore, we configure the `memtier_benchmark` tool to generate an access pattern that confirms Gaussian Distribution. We pre-populate the Redis server to host 16×10^6 key-value pairs, each with 1 KB in size. The memory footprint is about 16 GB. We then issue 3×10^6 queries against Redis while varying the read-write ratio in the workload. The results are shown in Figure 16. Linux and On-Demand-Fork tail latency decreases as there are more reads in the workload, which results in fewer page faults overall. Snappy 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. Both On-Demand-Fork and Snappy outperform Linux on maximum latency, as the snapshot call impacts this metric the most.

To summarize, Snappy significantly reduces the tail query latency for in-memory DBMS that leverage virtual memory for snapshotting compared to using `fork`.

7.4 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 [37] 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 1-minute warm-up after loading. We report average throughput during the two minutes 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 Providers and 56 worker threads for LeanStore.

Out-of-Memory Workloads. For out-of-memory scenarios (Figure 17), Tabby is much more resource-efficient. To show this, we normalized the throughput of these systems by the number of effective CPU cores required during the experiments. vmcache/LeanStore are 3.5×/2.5× worse regarding throughput per core. vmcache spends about 70% of the CPU cycles in the kernel for TLB-shootdown. For LeanStore, the pool of eviction threads constantly scans the buffer pool for cleaning pages to keep enough free frames for worker threads. Similar patterns were observed for skewed workloads. Compared to vmcache, Tabby removes the kernel overhead of manipulating virtual memory. Compared to LeanStore, Tabby does not have dedicated eviction threads, as all the evictions happen on the worker threads.

In-Memory Workloads. When data fits in memory, Tabby, vmcache perform similarly on read-only workloads (Figure 18a and Figure 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 18c), we observe similar patterns. However, when the access pattern is skewed (Figure 18d), all systems experience write contention and drop throughput 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.

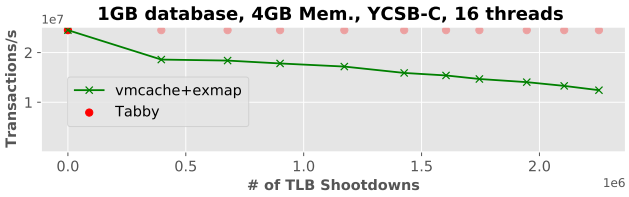


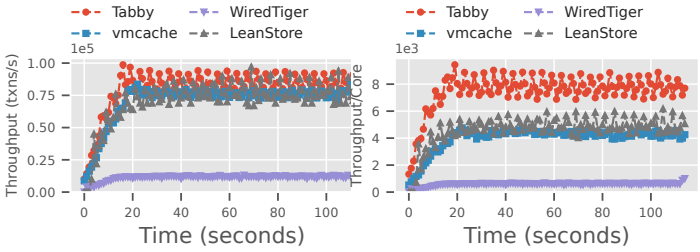
Figure 19: Performance Impact of TLB-shootdown on Cache Efficiency of In-Memory Workloads for vmcache+exmap.

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 CPU Efficiency. Next, we examine the performance impact of TLB shootdowns on CPU cache efficiency when the virtual memory (VM) subsystem is heavily manipulated. We simulate scenarios in which some CPU cores operate entirely in memory where accesses are skewed towards a small set of data records, while others heavily manipulate the exmap VM subsystem to replace buffer frames due to buffer pool miss (e.g., large scans or vacuum operations). The results, shown in Figure 19, demonstrate that TLB shootdowns can have up to a 2× impact on in-memory cores, even though these cores are independent of the I/O cores. This impact arises from the cycles spent on every CPU trapping into the kernel interrupt handler for TLB flushes, which leads to TLB cache misses on later instructions. However, this does not affect Tabby, as there are no TLB shootdowns when manipulating page tables due to our tighter co-design.

TPC-C Workload. We next evaluate the buffer pool systems using a more realistic workload - 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. Similar to the YCSB benchmark presented in the previous section, we use 64 threads to run the workload. We run the workload for 2 minutes and report the average throughput over time. The results are shown in Figure 20. Tabby, vmcache, and LeanStore are all able to saturate SSD, achieving similar throughput. They outperform WiredTiger by about 7×. Efficiency-wise, shown in Figure 20b, Tabby stands out and consistently outperforms vmcache and LeanStore by about 2× in throughput per core. 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 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 the TLB-shootdown elimination is the most important optimization, and one needs to pay attention to increased TLB-shootdown overheads when running as a privileged process.



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

Figure 20: TPC-C Throughput over Time (16 GB Buffer Pool, 64 Threads, 512 Warehouses≈100 GB)

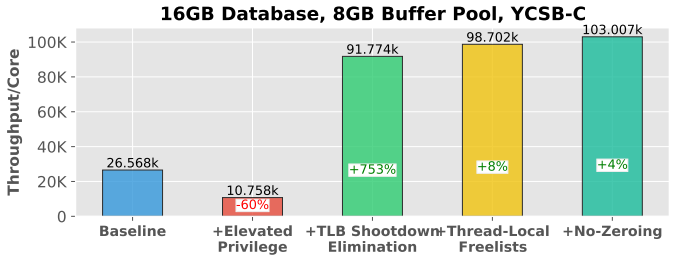


Figure 21: Performance Drill-down on Tabby

8 RELATED WORK

There is an extensive body of work in the area of DB-OS co-design. In this section, we discuss works not covered in Section 2.

DB-OS Co-Design. The DBOS project [65] focuses on building a cluster OS using scalable distributed databases [43, 48, 68, 77], for better resource management, observability, serverless application development [46], and debugging [51]. Currently, the DBOS project still runs DBMS in the userspace of a Linux kernel. Our co-design approach allows the DBMS in the DBOS project more control over the hardware. MxKernel [57] is a runtime system that advocates run-to-completion execution rather than OS threads for database systems. COD [35] 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.

Bypass Mechanisms for I/O Path. Data copying between user space and kernel has imposed significant CPU overhead on applications intended to fully exploit modern storage and network devices [40, 74]. One way of addressing the CPU overhead is to leverage kernel bypassing libraries such as DPDK and SPDK [73]. Therefore, many works [8, 11, 26, 42] focus on kernel bypass approach to build efficient user-space I/O stacks. These libraries implement drivers for I/O devices in user space and employ polling to process I/O requests at high speed, which wastes CPU cycles at idle time [31]. Conversely, it is also possible to reduce the overhead of system calls and copying by bypassing user space. A line of work exploits in-kernel computation (e.g., eBPF [7]) to offload user-space work – such as DBMS caching, proxy, B-tree traversal – to kernel [23, 24, 76, 80]. While bypass mechanisms improve I/O

path efficiency, they cannot address the inefficiencies of security-sensitive subsystems such as virtual memory, for which our co-design approach provides a solution. Note that our co-design can also co-exist with all the bypass mechanisms.

9 CONCLUSION

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/efficiency improvement and tail latency reduction.

References

[1] [n.d.]. 2024 CrowdStrike-related IT Outages. https://en.wikipedia.org/wiki/2024_CrowdStrike-related_IT_outages

[2] [n.d.]. AWS Instance Types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html

[3] [n.d.]. Azure BareMetal Infrastructure. https://learn.microsoft.com/en-us/azure/baremetal-infrastructure/

[4] [n.d.]. Bare Metal Solution. https://cloud.google.com/bare-metal/docs

[5] [n.d.]. e-BPF Maps. https://ebpf.io/what-is-ebpf/#maps

[6] [n.d.]. e-BPF Verification. https://ebpf.io/what-is-ebpf/#verificatio

[7] [n.d.]. extended Berkeley Packet Filter. https://ebpf.io/

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

[9] [n.d.]. io_uring 122M IOPS. https://www.reddit.com/r/linux/comments/wdlfjz/io_uring_122m_iops_in_2u_with_80_of_the_system/

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

[11] [n.d.]. Seastar. https://seastar.io/networking/

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

[13] [n.d.]. The good, bad, and compromisable aspects of Linux eBPF. https://pentera.io/blog/the-good-bad-and-compromisable-aspects-of-linux-ebpf/

[14] [n.d.]. Using Large Pages in KVM. https://www.linux-kvm.org/page/UsingLargePages

[15] 2016. Unikernels are unfit for production. https://www.tritondatacenter.com/blog/unikernels-are-unfit-for-production.

[16] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[17] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged {CPU} features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 335–348.

[18] Eli Bendersky. 2018. Measuring context switching and memory overheads for Linux threads.

[19] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. {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)*. 195–212.

[20] Sharada Bose, Priti Mishra, Priya Sethuraman, and Reza Taheri. 2009. 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, 167–182.

[21] Trevor Alexander Brown. 2015. 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*. 261–270.

[22] Matthew Butrovich. [n.d.]. *On Embedding Database Management System Logic in Operating Systems via Restricted Programming Environments.* Ph.D. Dissertation. Carnegie Mellon University.

[23] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. 2022. 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*. 617–630.

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

[25] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. 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.*

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

[27] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[28] Dean De Leo and Peter Boncz. 2019. Packed memory arrays-rewired. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 830–841.

[29] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[30] Eric Deehr, Wen-Qi Fang, H Reza Taheri, and Hai-Fang Yun. 2015. 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, 156–172.

[31] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*. 120–127.

[32] Nabil El Ioini, Ayoub El Majjodi, David Hastbacka, Tomas Cerny, and Davide Taibi. 2023. Unikernels Motivations, Benefits and Issues: A Multivocal Literature Review. In *Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum* (Ludwigsburg, Germany) *(ESAAM '23)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/3624486.3624492

[33] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

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

[35] Jana Giceva, Tudor-Ioan Salomie, Adrian Schüpbach, Gustavo Alonso, and Timothy Roscoe. 2013. COD: Database/Operating System Co-Design.. In *CIDR*.

[36] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, Berlin, Heidelberg, 393–481.

[37] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. 1994. Quickly generating billion-record synthetic databases. In *SIGMOD*. 243–252.

[38] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. 2015. Queues {don't} matter when you can {JUMP} them!. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 1–14.

[39] Martin Grund, Jan Schaffner, Jens Krueger, Jan Brunnert, and Alexander Zeier. 2010. The effects of virtualization on main memory systems. In *Proceedings of the sixth international workshop on data management on new hardware*. 41–46.

[40] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2090–2102.

[41] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards Buffer Management with Tiered Main Memory. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[42] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. {mTCP}: a Highly Scalable User-level {TCP} Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 489–502.

[43] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB*. 1496–1499.

[44] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE (ICDE)*. 195–206.

[45] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. {OSv—Optimizing} the Operating System for Virtual Machines. In *2014 usenix annual technical conference (usenix atc 14)*. 61–72.

[46] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, et al. 2022. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework.

arXiv preprint arXiv:2208.13068 (2022).

[47] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. 2021. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 376–394.

[48] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment* 5, 12 (2012).

[49] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

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

[51] Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2023. R3: Record-Replay-Retroaction for Database-Backed Applications. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3085–3097.

[52] Chih-En Lin. [n.d.]. [PATCH v4 00/14] Introduce Copy-On-Write to Page Table. https://lore.kernel.org/lkml/CANOhDtU3J8SUCzKtKvPPPrUHyo+LV5npNObHtYP_AK4W3LomDw@mail.gmail.com/T/.

[53] Chih-En Lin. [n.d.]. [RFC PATCH 0/6] Introduce Copy-On-Write to Page Table. https://lore.kernel.org/lkml/20220521040301.GA1508050@strix-laptop/T/.

[54] Anil Madhavapeddy and David J Scott. 2013. 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), 30–44.

[55] Umar Farooq Minhas, Jitendra Yadav, Ashraf Aboulnaga, and Kenneth Salem. 2008. Database systems on virtual machines: How much do you lose?. In *2008 IEEE 24th International Conference on Data Engineering Workshop*. IEEE, 35–41.

[56] Tobias Mühlbauer, Wolf Rödiger, Andreas Kipf, Alfons Kemper, and Thomas Neumann. 2015. High-performance main-memory database systems and modern virtualization: Friends or foes?. In *Proceedings of the Fourth Workshop on Data analytics in the Cloud*. 1–4.

[57] Michael Müller and Olaf Spinczyk. 2019. Mxkernel: rethinking operating system architecture for many-core hardware. In *9th Workshop on Systems for Multi-core and Heterogenous Architectures*.

[58] Pu Pang, Gang Deng, Kaihao Bai, Quan Chen, Shixuan Sun, Bo Liu, Yu Xu, Hongbo Yao, Zhengheng Wang, Xiyu Wang, et al. 2023. Async-fork: Mitigating Query Latency Spikes Incurred by the Fork-based Snapshot Mechanism from the OS Level. *arXiv preprint arXiv:2301.05861* (2023).

[59] Niklas Riekenbrauck, Marcel Weisgut, Daniel Lindner, and Tilmann Rabl. 2024. A Three-Tier Buffer Manager Integrating CXL Device Memory for Database Systems. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 395–401.

[60] Felix Schuhknecht. [n.d.]. Taking the Shortcut: Actively Incorporating the Virtual Memory Index of the OS to Hardware-Accelerate Database Indexing. ([n. d.]).

[61] Felix Schuhknecht and Justus Henneberg. 2022. Towards Adaptive Storage Views in Virtual Memory. *arXiv preprint arXiv:2209.01635* (2022).

[62] Felix Schuhknecht and Justus Henneberg. 2023. Accelerating Main-Memory Table Scans with Partial Virtual Views. In *Proceedings of the 19th International Workshop on Data Management on New Hardware*. 89–93.

[63] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired user-space memory access is possible! *Proceedings of the VLDB Endowment* 9, 10 (2016), 768–779.

[64] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting. In *Proceedings of the 2018 International Conference on Management of Data*. 245–258.

[65] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, et al. 2021. DBOS: a DBMS-oriented Operating System. (2021).

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

[67] Michael Stonebraker. 1981. Operating system support for database management. *Commun. ACM* 24, 7 (1981), 412–418.

[68] Michael Stonebraker, Sam Madden, and Pradeep Dubey. 2013. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *SIGMOD Rec.* 42, 1 (May 2013), 44–49.

[69] The gVisor Authors. [n.d.]. gVisor: The Container Security Platform.

[70] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[71] Wikipedia. [n.d.]. Pick Operating System. https://en.wikipedia.org/wiki/Pick_operating_system

[72] Wikipedia. [n.d.]. Transaction Processing Facility. https://en.wikipedia.org/wiki/Transaction_Processing_Facility

[73] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 154–161.

[74] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. 2021. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 195–211.

[75] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-demand-fork: A microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 540–555.

[76] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. 2022. {XRP}:{In-Kernel} Storage Functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 375–393.

[77] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2022. Lotus: scalable multi-partition transactions on single-threaded partitioned databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2939–2952.

[78] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2023. Two is better than one: The case for 2-tree for skewed data sets. *memory* 11 (2023), 13.

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

[80] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. 2023. Userspace Bypass: Accelerating Syscall-intensive Applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 33–49.