Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware

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

Christian Dietrich
Technische Universität Braunschweig
dietrich@ibr.cs.tu-bs.de

ABSTRACT

This paper explores the intersection of operating systems and database systems, focusing on the potential of specialized kernels for cloud-native database systems. Although the idea of custom, DBMS-optimized OS kernels is old, it is largely unrealized due to the demands of hardware compatibility and the reluctance of users to install specialized operating systems. However, the cloud and the database-as-a-service model make custom OS kernels realistic for the first time. Among specialized OS kernel architectures, unikernels stand out for relying on a single address space, eliminating the need for costly process isolation that is provided by general-purpose operating systems. They offer benefits such as the elimination of system call overhead, direct access to hardware, and reduced complexity. Beyond these immediate advantages, unikernels offer a unique opportunity: the possibility to revisit dated POSIX APIs. By allowing direct interaction with modern hardware primitives, unikernels pave the way for the development of novel abstractions that are not confined to the limitations of older APIs, opening doors to a new era of co-designed, high-performance cloud-native data processing systems and OS kernels.

PVLDB Reference Format:
Viktor Leis and Christian Dietrich. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. PVLDB, 17(8): 2115 - 2122, 2024.
doi:10.14778/3659437.3659462

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://doi.org/10.5281/zenodo.10672474.

1 INTRODUCTION

Operating Systems and Database Systems. In an OSDI 2021 keynote [50], Timothy Roscoe defined operating systems "software that multiplexes [a] machine's hardware resources, abstracts the hardware platform, and protects software principals from each other using the hardware". In essence, operating systems provide a uniform, abstract interface to the hardware and isolate different processes accessing the hardware. However, hardware abstraction and process isolation come at a cost, particularly for high-performance database systems that require fine-grained control over the hardware to optimize performance and resource utilization. As a result, it has long been recognized [21, 54] that custom OS kernels could offer more suitable OS APIs tailored to the needs of database systems and therefore better performance.

Why Have Custom OS-Kernels Not Been Successful? In practice, DBMS-optimized OS kernels never gained widespread adoption for two main reasons. First, few users would buy a DBMS that requires installing a custom OS as a prerequisite. Second, since users have different hardware setups, such a custom OS would have to implement and maintain drivers for a myriad of devices. Therefore, outside of narrow niche use cases, database systems simply had to adapt to the restrictions of general-purpose operating systems.

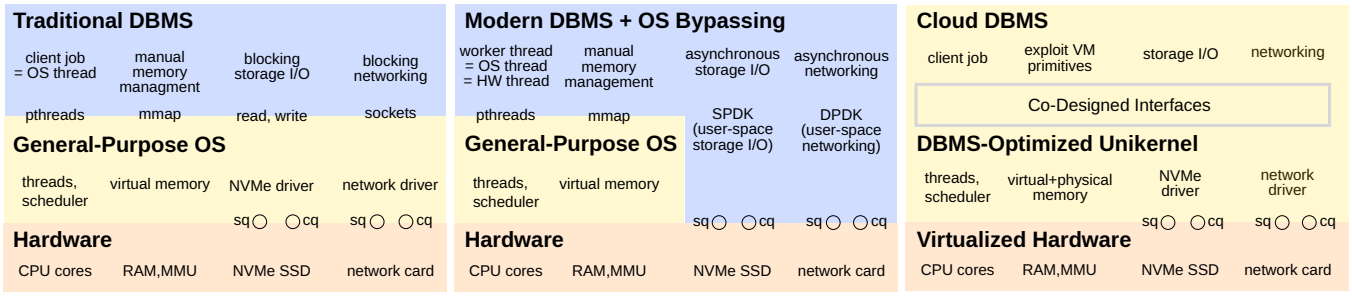
Why This Time Is Different: Cloud. The transition to the cloud has made the use of custom OS kernels in database systems more realistic. Instead of shipping DBMS software to customers for on-premise installation, database systems are increasingly offered as hosted services in public clouds. This shift means that custom kernels only need to be installed by the DBMS vendor, allowing users to benefit from a custom kernel without the burden of administration. Moreover, public clouds typically require support for only a handful of hardware devices. For example, AWS EC2 supports booting custom OS kernels, and a DBMS running on any recent EC2 instance only needs to implement two drivers: an AWS-specific EFA driver for networking and a generic NVMe driver for storage. This makes custom kernels a realistic option for cloud-native DBMSs.

OS Architecture Continuum. OS architectures vary in their degree of isolation: Microkernels [37] maximize isolation by segregating OS components (e.g., file systems) into separate processes. Monoliths, e.g., Linux or Windows, only distinguish between user and kernel space, executing bug-prone drivers [10, 48] with supervisor privileges. Container-based virtualization [40, 41] also falls into the monolith category, as it relies on namespace-isolated heavy-weight processes. Library OSes [2, 19] and the cloud-targeted *unikernels* [29, 31, 38], which are the focus of this work, co-locate all components into one address space without privilege isolation.

Kernel Integration. We argue that unikernels are an ideal foundation for cloud-native database systems as their lack of isolation allows for deep *DBMS/kernel integration*. With a unikernel, any thread can directly access any kernel data structure and hardware device. Hypervisor-based virtualization, where each VM runs a single service, ensures tenant isolation in the cloud. For many cloud services, the overhead of running a general-purpose OS, like Linux, is redundant, as there are no other processes or users to isolate from. Unikernels, therefore, eliminate this unnecessary layer of isolation. Section 3.1 discusses security implications in more detail.

Unikernel Benefits. For high-performance database management systems, unikernels offer several compelling advantages. (i) There is no (cost for) privilege-level transitions. (ii) Applications can directly control hardware *and* manipulate virtual-memory page tables. (iii)

Proceedings of the VLDB Endowment, Vol. 17, No. 8 ISSN 2150-8097.
doi:10.14778/3659437.3659462



(a) Traditional division of labor (b) Modern DBMS with (optional) bypassing (c) Our vision for DBMS/OS co-design

Figure 1: Models of DBMS/OS interaction

Isolation is a major source of OS implementation complexity; hence, unikernels can be smaller, simpler, and more predictable. (iv) Due to their simplicity, unikernels boot more quickly than general-purpose kernels.

New Abstractions. Although the benefits listed above are significant, we believe the most profound advantage of unikernels lies in their ability to move beyond the decades-old POSIX API. Existing OS APIs severely limit the design space: for example, using `mmap` it is not possible to implement ARIES-style write-ahead logging with in-place writes [11]. Unikernels make it both straightforward and efficient to develop new abstractions based on the primitives provided by modern hardware such as the page tables, TLB, and inter-processor interrupts. For example, the virtual-memory page table becomes a simple radix data structure that can be manipulated directly instead of having to rely on expensive and functionally limited system calls. The co-design of data processing systems and OS kernels offers the unique opportunity to develop new abstractions without being restricted to APIs that were designed for single-core systems with scarce memory.

2 OPERATING SYSTEMS AND DATABASES

Before we present specific ideas enabled by co-designing database systems and unikernels in Section 3, let us first discuss why the current state of operating systems falls short for high-performance data processing systems.

Traditional Abstractions. A main task of an operating system kernel is to manage hardware. Specifically, an OS provides a uniform abstraction over hardware and multiplexes it among different users. The four main hardware resources most relevant for database systems are CPU cores, main memory, storage, and networking. In the traditional model, the OS provides the abstraction *threads of execution*, which the OS scheduler dynamically maps to CPU cores. Storage and network I/O are done through synchronous, blocking system calls. When a thread blocks on an I/O operation, it is not scheduled until the device signals completion via an interrupt. In this model, the OS controls all scheduling decisions and receives all hardware signals.

Traditional Abstraction Worked Well. As Figure 1(a) illustrates, a straightforward way to implement a DBMS on top of the traditional OS abstraction is to have one thread (or process) for each client connection. Each thread listens on a network socket for incoming requests using a blocking system call. If a request arrives, the OS schedules the thread for execution on some CPU core. The thread can then process the query and eventually return the result to the client through another blocking socket system call. During query execution, buffer-pool page misses may result in storage I/O system calls and therefore additional context switches. This model is easy to use, conceptually elegant, and provides a clean separation of concerns between the DBMS and OS. The DBMS relies heavily on the abstractions provided by the OS. Through blocking system calls (and, when needed, preemptive multitasking) the OS is in full control and can ensure that all hardware resources are well utilized. In a world where (disk and network) I/O was relatively slow in comparison to the CPU, the model was efficient due to negligible context switching overhead. Consequently, this synchronous model has historically been very successful, remaining the basis for widely-used systems like PostgreSQL.

Traditional Abstraction On Modern Hardware. Unfortunately, on modern hardware, the synchronous model runs into severe performance problems. Modern storage devices are very fast, which means that the kernel CPU overhead of I/O stack CPU can be substantial[1]. Storage devices are also highly parallel: a single modern SSD, for example, is capable of simultaneously executing on the order of 100 I/O requests and over one million requests per second. Consequently, it is nearly impossible to exploit modern storage devices using blocking I/O system calls [23]. A second, more conceptual, problem with the traditional model is that it does not take intra-query parallelism into account, which is crucial on modern hardware with dozens of CPU cores. Intra-query parallelism is challenging because it breaks the one-to-one mapping between client connections and threads.

Asynchronous I/O and Worker Thread Scheduling. To address the challenges associated with synchronous I/O and intra-query parallelism, modern high-performance database systems therefore schedule I/O requests and CPU cores themselves. To do this, they launch as one worker thread per CPU core [33], rely on asynchronous I/O interfaces such as io_uring [23], and avoid OS services such as software RAID and the file system [22, 46]. In this model, the database system has to decide how many threads to use for each query, which means that it requires a scheduler that ideally considers other currently-running queries and whether they are I/O or CPU-bound. The DBMS also has to manage I/O requests, with the

[1] On Linux, every I/O operation takes on the order of 10,000 CPU cycles.

OS becoming a mere intermediary passing asynchronous requests to asynchronous hardware devices. Note that these scheduling and I/O decisions used to be the main purpose of an OS. On modern hardware, the only way to achieve good performance is through low-level OS interfaces, which push most of the responsibilities to the application. Thus, the OS cannot fulfill its traditional role as coordinator between the application and the hardware.

Kernel Bypassing. The high bandwidth of modern I/O devices causes substantial CPU load just for moving data to the CPU. For example, two recent studies showed that exploiting the bandwidth of eight NVMe SSDs [23] or of one 100 Gbit Ethernet NIC [17] requires roughly half the CPU cores – despite using the state-of-the-art io_uring interface. This has motivated the trend to bypass the OS using user-space libraries such as DPDK (for networking) and SPDK (for storage I/O). This OS bypassing approach is illustrated in Figure 1(b); it can reduce the CPU load in high-throughput situations considerably [14, 23]. However, user-space I/O relies on polling rather than interrupts, for which Linux provides no means to forward device interrupt requests (IRQs) as asynchronous events to the user space. When the request rate is low, polling wastes CPU cycles and leads to unnecessary power consumption because it prevents CPU cores from reducing their clock frequency. Data-plane OSes, such as IX [8], Arrakis [49] or the demikernel [61], offer a generalized approach to kernel-bypass APIs. However, they rely on hardware features not available in virtualized guest OSes (SR-IOV [8, 49]) or aim to keep the problematic [4, 6, 11, 18, 56, 58] POSIX API [61]. Instead of kernel bypass, in this work, we aim for *kernel integration*.

Conclusion. Despite being elegant, traditional synchronous OS abstractions cannot achieve good performance on modern hardware. High-performance database systems are therefore forced to implement CPU and I/O scheduling themselves. This can even go so far as bypassing OS device drivers and higher-level abstractions altogether through user-space I/O. However, if the only way of getting good performance is to do everything in the application, *why use a general-purpose OS anyhow?*

3 UNIKERNELS FOR DATABASES

Unikernels are lightweight operating systems specifically designed for the cloud. The main idea is to compile the application and the OS kernel into one system image running in a hypervisor-based virtualized environment. The hypervisor allows the application to run in kernel mode and execute in the same memory address space as the kernel, as Figure 1(c) illustrates. As we mentioned in the introduction, this property makes unikernels simple (tens of thousands of lines of code instead of millions) and efficient (no system call overhead, no process/user isolation cost). The unikernel approach was pioneered by the Mirage system [39], which relied on the OCaml programming language.

3.1 What Unikernels Offer

OSv. Later unikernels such as OSv [29] and Unikraft [31] implement the standard Unix API POSIX, making them programming-language agnostic. We chose OSv (rather than the more recent Unikraft) due to its good multi-core support and its clean, readable, and efficient C++ code. OSv supports x86_64 and arm64 as well as multiple virtualization technologies such as QEMU/KVM, Xen, and Firecracker [1]. During development, unikernel-based deployments are debugged using standard debuggers like gdb. In this setting, the debugger runs on the host OS and the developer can easily inspect the application *and* kernel code. In production, OSv can be executed either using a KVM-virtualized EC2 instance or using a bare-metal EC2 instance running Firecracker. The former might be suitable for handling the base load, while the latter for handling fluctuating workloads in a function-as-a-service-like way.

POSIX Versus New Abstractions. OSv runs many applications out-of-the-box or with only minimal modifications [47]. However, because POSIX is large and includes many obscure features, all unikernels implement only a subset of POSIX. Indeed, it would *not* be a good idea to be 100% compatible with Linux, since this would destroy the simplicity of unikernels. Instead, we argue for co-designing the DBMS with the unikernel by implementing novel DBMS-specific abstractions on virtualized cloud hardware. Only the simplicity of unikernels makes this approach realistic. For example, the entire virtual memory subsystem of OSv is located in one 2,100 line C++ file (core/mmu.cc). For comparison, the corresponding code in Linux (mm/*) is over 110,000 lines of C code. Most of this subsystem implements isolation or obscure features that are not relevant for database systems.

Unikernels Offer New Hardware Primitives. Running the DBMS in kernel mode offers direct access to the (virtualized) hardware and primitives that Linux, due to its shared-machine model, does not expose to the user space. As demonstrated by DUNE [7], direct and efficient access to these primitives brings significant performance opportunities. A unikernel-based DBMS can exploit such opportunities while remaining compatible with virtualized cloud environments. Specifically, we believe that the following hardware primitives can be exploited by database systems:

- CPU: preemption primitives, sleep/power states
- Virtual Memory: page table and TLB manipulation
- Hypervisor: memory ballooning and CPU hot-plugging
- I/O: direct access to submission and completion queues, interrupt control, device configuration

Unikernel Security. Due to their simplicity, unikernels have a vastly reduced attack surface [55]. However, they may lack standard security measures (e.g., address and privilege isolation) and common hardening techniques (e.g., guard pages, address space layout randomization) [55]. While the latter are mere implementation deficits, the former is not problematic in cases where a unikernel hosts a single tenant. After all, the big prize for an attacker is obtaining illegal access to the database, and escalating a user-space vulnerability to kernel space brings limited additional benefits. Thus, database security primarily depends on application/DBMS security rather than the isolation traditional OSes can provide. Moreover, it is possible to introduce hardware-assisted isolation, such as Intel MPK for thread or range isolation [35], and privilege isolation without address virtualization [24, 26]. Finally, widely-used OS kernels are not free of security bugs. For example, due to recurring kernel exploits, Google has disabled io_uring in ChromeOS, on Android, and on production servers [30].

3.2 Compute Scheduling Opportunities

User-Space Tasks Have Problems. As discussed in Section 2, modern database systems are forced to schedule CPU tasks and I/O operations themselves if they want to fully exploit modern hardware. For example, LeanStore uses core-pinned worker threads that maintain a queue of user-space tasks [23]. When a task has a page miss, it submits an asynchroneous I/O request, enqueues itself, and then switches to another task (using user-space stack switching). Although this approach is efficient, it effectively re-implements the OS scheduler in user space and is not robust for two reasons. First, a CPU-intensive task may monopolize the CPU core because user-space schedulers have no access to the raw preemption mechanism (i.e., hardware timers, cross-core activations). Therefore, preemptive user-space scheduling has to rely on extensive and relatively inefficient kernel support [3, 25] to hide the preemption machinery or it has to emulate a bare-metal environment on top of expensive OS abstractions (i.e. POSIX signals [42]). Second, if the user-space task unintentionally blocks, the core is temporarily wasted as the OS is not aware of other waiting tasks.

Unikernel Threads Are Lightweight and Offer Preemption. User-space scheduling is only necessary because kernel-level threads are expensive. A context switch in Linux takes at least 1us [9], while switching between stacks in user space takes only 10ns [23]. In unikernels, there is no distinction between user-space tasks and kernel-level threads; threads can suspend themselves, manipulate the run queue on another core, block preemption, or interleave their execution like co-routines. At the same time, the OSv scheduler has a full picture of all threads and can preempt each of them if they monopolize the CPU. Getting rid of OS-invisible user-space tasks avoids scheduling gaps, makes all logical DBMS jobs visible, and the DBMS can track system-resource utilization.

DBMS-Aware Scheduling. Once the scheduler gains control and visibility over tasks, we can also tackle the intra-query parallelism problem raised in Section 2. Currently, the DBMS has to decide how many threads to use for each query. However, we argue that it would be better if the scheduler could ask logical jobs to parallelize themselves. In this design, all required information for good scheduling decisions (CPU load, I/O utilization, job priorities, etc) is available in one place and cheap to obtain. In a unikernel, such a scheduler is much easier to implement because context switches are cheap and we get the capability of preemptively interrupting threads. In Section 4, we show how ad-hoc work distribution can speed up otherwise competing operations.

3.3 Virtual Memory Opportunities

Using Virtual Memory For Functionality, Not For Isolation. General-purpose operating systems primarily use the virtual-memory hardware (e.g., MMU and TLB) to separate the user space and kernel space and to isolate processes from each other. Without the need for isolation, unikernels can provide direct control over the virtual-memory hardware and enable new use cases. Specifically, we see three opportunities for database systems.

Caching. Virtual memory can be exploited to implement highly efficient buffer management by using the virtual memory page table as hardware-assisted indirection table. In earlier work [32], we showed that this can be implemented in Linux using either slow, existing system calls (*vmcache*) or by extending Linux with faster system calls (*exmap*). With a unikernel, implementing such a caching design becomes both faster and simpler.

VM-Aware Algorithms and Data Structures. Virtual memory (VM) has also been used to implement snapshotting [28, 52], dynamic data structures [34, 51], and variable page sizes [32, 44]. Despite their conceptual elegance, these proposals have not been widely adopted. We believe that a major reason for this is that VM primitives are slow and do not scale in Linux [11, 32]. For example, using its `pagemap` interface to check if a random 4 KiB page is present within a 4 GiB area takes 1.8 us–4.8 us on a 16-core CPU; with OSv it takes 40 ns–44 ns. An alternative to reduce the overhead of VM operations is to use huge (2 MiB) pages, and we plan to support both page sizes. However, huge pages are not always beneficial for OLTP workloads with random access patterns. Scalable and efficient 4 KiB VM operations therefore enable more use cases.

Memory Allocation. Another use case for VM is memory allocation for intermediate query processing results, which occur at a high rate for in-memory query processing [16] and often require large contiguous VM ranges (e.g. for hash tables). For such allocations, existing memory allocators have two options: They can either directly pass each (de)allocation to the OS with *mmap()* and *munmap()*, which causes on-demand page faults after every new allocation. With Linux, such page faults scale poorly: Installing a 4 KiB page with 1 thread takes 1.05 us and 4.17 us with 16 threads. General-purpose allocators, such as jemalloc, therefore may keep the memory in-process, which poses the risk of memory fragmentation due to variable allocation sizes. In unikernels, option A becomes more attractive as the huge VM address space ($\geq$ 256 TiB) eases fragmentation and page-faulting is faster and potentially more scalable: a lock-free page-fault fast path that installs a preallocated frame takes between 0.5 us (1 thread) and 1.29 us (16 threads).

3.4 Hypervisor Opportunities

Giving the DBMS access to lower levels does not have to stop at the kernel level; paravirtualization [5] extends it to the hypervisor. For example, the hypervisor can reclaim overcommitted memory by manipulating the OS state without guest interaction [57]. A DBMS could even expose its memory allocator state to the hypervisor, allowing for more efficient memory ballooning and therefore more effective memory over-provisioning. And with para-virtualized CPU hot-plugging, the DBMS can directly request additional CPU cores as needed [60]. Allowing the DBMS to exchange information with the hypervisor and dynamically change memory and CPU allocations can thus be used to improve resource utilization.

3.5 I/O Opportunities

NVMe Storage. Today, almost all storage devices use the standardized NVMe protocol, which means that only one driver is needed for storage. NVMe is also quite simple: we implemented a basic prototype driver supporting reads and writes in several hundred lines of code. NVMe is based on queues that can be accessed by both the host CPU and the storage device. After allocating one or more submission and completion queues and registering them with the device, submitting operations merely involves writing to the submission queue. Once an I/O operation is finished, it will appear in

the completion queue, and through DMA the data will appear at the desired location in memory. It is noteworthy that this mirrors the asynchronous, queue-based I/O model that modern OS interfaces, such as io_uring, employ. Basically, io_uring merely provides an additional queue on top of the NVMe queue that causes substantial CPU overhead [23]. A unikernel-based design can simply expose NVMe queues to the DBMS, bypassing this unnecessary layer.

Networking. Unfortunately, network cards are not as standardized as storage devices. In AWS EC2, all modern instances use the Elastic Network Adapter (ENA), on which we plan to initially focus. Another challenge with networking is that in addition to a low-level packet interface, database systems generally also require support for TCP, e.g., by using a user-space implementation of TCP [27]. An alternative to TCP would be to rely on Amazon's Elastic Fabric Adapter (EFA) [62], which supports the Scalable Reliable Datagram (SRD) protocol. This is a packet-based *reliable but unordered* protocol. Packets sent will eventually arrive, but not necessarily in the same order they were sent. We believe that co-designing the DBMS communication stack with SRD could be a very good way to avoid the CPU overhead of TCP.

Interrupts and Power Management. Achieving maximum performance with modern storage and network requires polling rather than interrupts-based I/O [14, 23]. The problem with polling is that it becomes very wasteful in terms of power consumption when the event rate is low. In low-throughput scenarios, it is better to switch to interrupts and disable polling. In contrast to kernel-bypassing approaches, unikernels allow enabling, disabling, and routing interrupts dynamically depending on the workload.

4 EVALUATION: VIRTUAL MEMORY

The goal of this section is to demonstrate one specific example of DBMS-unikernel co-design. We focus on virtual memory snapshotting, comparing the Linux VM subsystem with a tightly-integrated OSv implementation.

Benchmark Setup. We conduct our benchmarks within a virtual machine with 16 cores and 12 GiB of DRAM, which is sufficient physical memory for all experiments. We disabled memory ballooning to avoid unpredictable memory-access slowdowns caused by hypervisor-level fragmentation. Within the VM, we used Linux (v6.1.0, Debian Unstable) or OSv (`8c792811d`) as operating systems. We execute the VM on top of a physical machine with an AMD EPYC 9554P processor (64 cores, 128 HW threads, 384 GiB DRAM, 1 NUMA domain) and used QEMU (v8.0.2) with hardware-assisted virtualization (KVM). Our modified OSv version, the used benchmarks, and the resulting data is available [15].

Copy-on-Write Memory Snapshots. Fast snapshots of memory regions are a useful primitive, for example, to separate read-only OLAP queries from OLTP transactions. The original design of HyPer [28] used the `fork()` system call as a snapshotting mechanism. More fine-grained snapshotting variants have been proposed as well [52]. In both cases, the key idea is to leverage the OS' capability to create a consistent copy-on-write snapshot of a process for an OLAP job. Although fork-based snapshotting was the key original idea behind HyPer [28], due to involved OS overheads and lack of control the idea was eventually abandoned in favor of software-based MVCC [45]. It may be time to revisit this idea.

Microbenchmark. For our scenario, we allocate a 4 GiB anonymous memory mapping on which n OLTP threads execute random updates using atomic fetch-and-add operations. Every 3 seconds, we initiate a concurrent OLAP job, which creates a read-only copy-on-write snapshot and subsequently scans over the snapshot, accumulating 32-bit integers; afterward, it destroys the snapshot. As the OLAP job runs, the OLTP threads continue to manipulate the primary buffer, inducing frequent page faults to resolve established copy-on-write mappings with an actual copy. In this benchmark, it is likely that all copy-on-write (CoW) mappings are resolved before the OLAP job finishes. We show the results as copy, scan, update, or destroy throughput in GiB per second in Figure 2.

Linux. For Linux, we use `fork()` to create the snapshot and run the OLAP job single-threaded in a separate process. Linux establishes 60 GiB of copy-on-write mapping per second, while snapshot destruction is six times slower due to the poor scalability of Linux' buddy page-frame allocator [59]. While the OLAP process runs, the OLTP threads achieve a 89 percent lower throughput as Linux performs one TLB shootdown per CoW page fault.

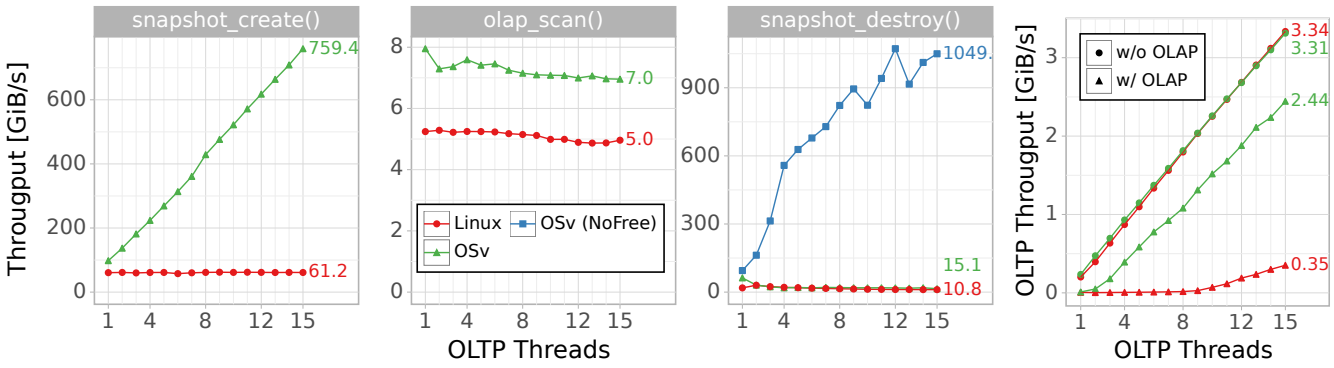
OSv. As unikernels, by design, do not support `fork()`, we extend OSv with a copy-on-write snapshot primitive, inspired by `snapshot_vma()` [52], and apply it to the OLTP/OLAP scenario. Furthermore, we introduce two further techniques that highlight the possibilities of our co-design approach: parallel snapshots (compute opportunity, Section 3.2) and reader-side TLB invalidation (memory opportunity, Section 3.3).

Parallel Snapshotting. With *parallel snapshots*, we speed up the creation of snapshots. In Linux, the OLTP threads compete with the snapshot creation for page-table locks and induce frequent TLB shootdowns, slowing down the snapshot request of the OLAP thread. In OSv, during snapshot creation, we let OLTP threads that trigger a CoW fault assist the OLAP thread with the creation process. In this process, OLAP and OLTP threads share the page-table writer lock and coordinate their page-table copying with atomic instructions. In a general-purpose OS, such a writer-lock sharing and CPU-time donation from the page-fault context is hard to implement and undesirable.

Reader-Side TLB Invalidation. As a second technique, we introduce *reader-side TLB invalidation*. For snapshotted regions, we removed the global TLB shootdown from the page-fault handler, which is usually required to inform other cores about a resolved CoW mapping to avoid stale reads. In OSv, the DBMS runs in supervisor mode, which allows readers to proactively refresh the TLB before accessing a snapshotted page. In our benchmark, the OLTP threads perform a single-page TLB invalidation for every transaction while a snapshot exists. In a DBMS, one could place this TLB invalidation in the buffer manager's `fix()` operation.

Results. In Figure 2, we see that our OSv-based implementation outperforms Linux: (i) Creating snapshots is faster as more cores help to copy the page tables. (ii) OLAP scan and the OLTP transactions are faster as they are not interrupted by frequent TLB shootdowns. (iii) Snapshot destruction is faster as we have to perform less bookkeeping compared to Linux (i.e., no reverse mapping).

Allocator Optimization. To further improve on snapshot destruction we also tried to parallelize this operation. However, the contention in OSv's frame allocator even reduced the throughput. Therefore, we created a *NoFree* variant that removes the allocator



(a) Phases of an OLAP Job on an Copy-on-Write Snapshot **(b) Impact on OLTP Operations**

Figure 2: Snapshot Copy-On-Write Benchmark

from the measurement and only collects freed frames in an array, similar to the process-local memory pools of ExMap [32]. If assisted by all OLTP threads, snapshot destruction now scales linearly with the number of cores, becoming 96 times faster than Linux.

Memory Bandwidth. Page tables are 1/512 of the VM space. Hence, the create and destroy primitives process page tables at 1.5 and 2 GiB/s respectively. Due to reference counter updates inducing a random-access pattern, this falls below the peak memory bandwidth (~395 GiB/s). Hiding this latency with prefetching could accelerate these VM operations significantly. Hence, although we already outperform Linux, the performance of VM snapshotting can be improved even further, e.g., using prefetching.

Summary. Using snapshotting as an example use case, we showed that (modified) unikernels enable superior performance for manipulating virtual memory. The optimizations described in this section were implemented in less than 1,000 lines of code.

5 RELATED WORK

The tension between the abstractions of general-purpose operating systems and the requirements of database systems has been observed already five decades ago [21, 54]. While there have been attempts at DBMS-specific operating systems, for example, as part the Gamma [13] project, almost all widely-used database systems rely on standard operating systems. More recent DBMS/OS co-design projects include DBOS [36, 53], MxKernel [43], and COD [20].

Like our proposal, **DBOS** focuses on the cloud, in particular on orchestrating distributed compute nodes. Currently, DBOS is based on Firecracker and it could be combined with a unikernel. We therefore believe that the ideas presented in this paper are complementary with the DBOS project.

MxKernel emphasizes the potential superiority of uninterruptible run-to-completion tasks over threads for executing query plans on heterogeneous architectures. In contrast, we will consider all important resources (CPU, memory, I/O) and aim for cloud-native DBMS system where the hardware pool is highly standardized.

From a high-level perspective, the **COD** proposal from 2013 [20] shares many of the goals of our work. COD emphasizes the need to "open up the OS" and design new declarative interfaces between the DBMS and the OS. Ten years later, the transition to the cloud and the existence of unikernels finally make this approach realistic. To the best of our knowledge, this is the first paper that specifically focuses on the advantages of unikernels for DBMS/OS co-design.

6 TOWARDS UNIKERNEL-DBS IN THE CLOUD

Unikernels in Cloud Data Planes. Many cloud-native systems are internally decomposed into several components. Snowflake, for example, consists of a multi-tenant control plane that performs management tasks and per-tenant data plane clusters that executes queries [12]. We do not strive to replace Linux for all components, but only for the performance-critical and resource-intensive data plane. In cloud-native systems, this data-plane layer is usually an elastic and resouce-intensive component. This makes the fast boot times of unikernels (<1s) highly useful. An integrated scheduling approach also allows for balancing the current resource demand with the cloud provider's pricing model.

Performance and Simplicity Through Novel Abstractions. Much of the complexity of today's high-performance data processing systems comes from having to explicitly manage modern hardware because the existing OS abstractions are not up to the task. In other words, database systems effectively already perform the traditional hardware management job of operating systems – but without having the low-level tools an OS has. Our vision of a unikernel-based cloud-native DBMS with a co-designed, hardware-centric, zero-cost hardware interface not only has the potential for higher efficiency, but it can also *simplify* crucial DBMS tasks such as CPU scheduling, memory management, and I/O. Over time, we envision the emergence of new unikernel-level abstractions, which may prove useful not just for database systems but also for other demanding applications such as high-performance computing and machine learning. Finally, let us mention that unikernels may also make it more efficient to exploit not just commodity hardware, but also accelerators such as FPGAs, TPUs, or DPUs.

ACKNOWLEDGMENTS

We thank the reviewers for their detailed and insightful feedback. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 501887536.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*.

[2] T.E. Anderson. 1992. The case for application-specific operating systems. In *WWOS*. https://doi.org/10.1109/WWOS.1992.275682

[3] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1992. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems* (1992). https://doi.org/10.1145/146941.146944

[4] Jens Axboe. 2019. *Efficient IO with io_uring*. https://kernel.dk/io_uring.pdf (Accessed on 09/04/2024).

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *SOSP*. https://doi.org/10.1145/945445.945462

[6] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. 2019. A Fork() in the Road. In *HotOS*. https://doi.org/10.1145/3317550.3321435

[7] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *OSDI*.

[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: a protected dataplane operating system for high throughput and low latency. In *OSDI*.

[9] Eli Bendersky. 2018. *Measuring context switching and memory overheads for Linux threads*. https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/ (Accessed on 09/04/2024).

[10] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. https://doi.org/10.1145/502034.502042

[11] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR*.

[12] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. https://doi.org/10.1145/2882903.2903741

[13] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. 1990. The Gamma Database Machine Project. https://web.eecs.umich.edu/~mozafari/fall2015/eecs584/papers/gamma.pdf. (1990). (Accessed on 09/04/2024).

[14] 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 *SYSTOR*. https://doi.org/10.1145/3534056.3534945

[15] Christian Dietrich and Viktor Leis. 2024. *Software/Data Artifact for "Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware"*. https://doi.org/10.5281/zenodo.10672474

[16] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. Experimental Study of Memory Allocation for High-Performance Query Processing. In *ADMS Workshop*.

[17] Domink Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *PVLDB* 16, 11 (2023). https://doi.org/10.14778/3611479.3611486

[18] Pekka Enberg, Ashwin Rao, Jon Crowcroft, and Sasu Tarkoma. 2022. Transcending POSIX: The End of an Era? *;login:* (2022).

[19] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP*. https://doi.org/10.1145/224057.224076

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

[21] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Advanced Course: Operating Systems (Lecture Notes in Computer Science, Vol. 60)*. Springer, 393–481. https://doi.org/10.1007/3-540-08755-9_9

[22] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[23] 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. *PVLDB* 16, 9 (2023). https://doi.org/10.14778/3598581.3598584

[24] Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. 2015. dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel. In *RTAS*. https://doi.org/10.1109/RTAS.2015.7108449

[25] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghost: Fast & flexible user-space delegation of linux scheduling. In *SOSP*. https://doi.org/10.1145/3477132.3483542

[26] NanoVMs Inc. 2023. *Nanos.org*. https://nanos.org/ (Accessed on 09/04/2023).

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

[28] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. https://doi.org/10.1109/ICDE.2011.5767867

[29] 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 *USENIX ATC*.

[30] Tamás Koczka. 2023. *Learnings from kCTF VRP's 42 Linux kernel exploits submissions*. https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html (Accessed on 09/04/2024).

[31] Simon Kuenzer, Vlad-Andrei Badoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Stefan Teodorescu, Costi Raducanu, Cristian Banu, Laurent Mathy, Razvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: fast, specialized unikernels the easy way. In *EuroSys*. https://doi.org/10.1145/3447786.3456248

[32] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *PACMOD* 1, 1 (2023). https://doi.org/10.1145/3588687

[33] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. https://doi.org/10.1145/2588555.2610507

[34] Dean De Leo and Peter A. Boncz. 2019. Packed Memory Arrays - Rewired. In *ICDE*. https://doi.org/10.1109/ICDE.2019.00079

[35] Guanyu Li, Dong Du, and Yubin Xia. 2020. Iso-UniK: lightweight multi-process unikernel through memory protection keys. *Cybersecurity* 3 (2020). https://doi.org/10.1186/s42400-020-00051-9

[36] Qian Li, Peter Kraft, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Jason Li, Michael J. Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2022. A Progress Report on DBOS: A Database-oriented Operating System. In *CIDR*.

[37] Jochen Liedtke. 1995. On μ-Kernel Construction. In *SOSP*. https://doi.org/10.1145/224057.224075

[38] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *ASPLOS*. https://doi.org/10.1145/2451116.2451167

[39] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David J. Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *ASPLOS*. https://doi.org/10.1145/2499368.2451167

[40] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *SOSP*. https://doi.org/10.1145/3132747.3132763

[41] Dirk Merkel et al. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* (2014).

[42] Malcolm S. Mollison and James H. Anderson. 2013. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *RTAS*. 283–292. https://doi.org/10.1109/RTAS.2013.6531100

[43] Jan Mühlig, Michael Müller, Olaf Spinczyk, and Jens Teubner. 2020. mxkernel: A Novel System Software Stack for Data Processing on Modern Hardware. *Datenbank-Spektrum* 20, 3 (2020). https://doi.org/10.1007/s13222-020-00357-5

[44] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[45] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*. https://doi.org/10.1145/2723372.2749436

[46] Lam-Duy Nguyen and Viktor Leis. 2024. Why Files If You Have a DBMS?. In *ICDE*.

[47] OSv. 2023. *OSv Applications*. https://github.com/cloudius-systems/osv-apps (Accessed on 09/04/2024).

[48] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. 2011. Faults in Linux: Ten years later. In *ASPLOS*. https://doi.org/10.1145/1950365.1950401

[49] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Transactions on Computer Systems* (2015). https://doi.org/10.1145/2812806

[50] Timothy Roscoe. 2021. It's Time for Operating Systems to Rediscover Hardware. https://www.usenix.org/conference/osdi21/presentation/fri-keynote. In *OSDI*. (Accessed on 09/04/2024).

[51] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *PVLDB* 9, 10 (2016). https://doi.org/10.14778/2977797.2977803

[52] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual

Snapshotting. In *SIGMOD*. https://doi.org/10.1145/3183713.3196904

[53] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael J. Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2021. DBOS: A DBMS-oriented Operating System. *PVLDB* 15, 1 (2021). https://doi.org/10.14778/3485450.3485454

[54] Michael Stonebraker. 1981. Operating System Support for Database Management. *CACM* 24, 7 (1981), 412–418. https://doi.org/10.1145/358699.358703

[55] Joshua Talbot, Przemek Pikula, Craig Sweetmore, Samuel Rowe, Hanan Hindy, Christos Tachtatzis, Robert Atkinson, and Xavier Bellekens. 2020. A Security Perspective on Unikernels. In *Cyber Security*. https://doi.org/10.1109/CyberSecurity49315.2020.9138883

[56] Chen Wang, Kathryn Mohror, and Marc Snir. 2021. File System Semantics Requirements of HPC Applications. In *HPDC*. https://doi.org/10.1145/3431379.3460637

[57] Yaohui Wang, Ben Luo, and Yibin Shen. 2023. Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Meta-data Management. In *USENIX ATC*. USENIX Association, Boston, MA, 769–783.

[58] Jinpeng Wei and Calton Pu. 2005. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study.. In *USENIX FAST*, Vol. 4. 155–167.

[59] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. 2023. LLFree: Scalable and Optionally-Persistent Page-Frame Allocation. In *USENIX ATC*.

[60] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, Calton Pu, and Hakan Hacgümüş. 2014. SmartSLA: Cost-sensitive management of virtualized resources for CPU-bound database services. *IEEE Transactions on Parallel and Distributed Systems* 26, 5 (2014), 1441–1451. https://doi.org/10.1109/TPDS.2014.2319095

[61] 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 *SOSP*. https://doi.org/10.1145/3477132.3483569

[62] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. 2022. EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?. In *DaMoN*. https://doi.org/10.1145/3533737.3538506