SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Towards DBMS-aware Memory Management in Unikernels

Martin Lindbüchl

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Towards DBMS-aware Memory Management in Unikernels

In Richtung DBMS-bewusstem Speichermanagement in Unikerneln

Author:	Martin Lindbüchl
Examiner:	Prof. Dr. Pramod Bhatothia
Supervisor:	Ilya Meignan–Masson
Submission Date:	18.03.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 18.03.2025 Martin Lindbüchl

Acknowledgments

I would like to thank Prof. Dr. Pramod Bhatothia for giving me the opportunity to explore this topic at his chair. I further want to express my gratitude to Ilya Meignan–Masson for his constant support and guidance during the last 4 months. Dedicating a lot of his time to answering my questions and providing feedback on my thesis and my code he contributed greatly to the quality of both.

Lastly, I would like to extend my thanks to my family and friends for their patience and support as I continuously shared thoughts about my work.

Abstract

The move towards cloud environments within recent years has increased flexibility of choice for the virtualized operating system (OS). This motivates to reconsider limitations posed by general-purpose OSs on database systems, including the mismatch between the abstractions provided by the Linux kernel and the functionality required for hardware-assisted, scalable memory management within the database management system (DBMS).

To overcome this issue, recent work proposes unikernels as lightweight library OS for DBMS/kernel integration. Unikernels are designed to host a single application in cloud environments, avoiding most of the complexity of general-purpose OSs.

However, current legacy unikernels are bound by POSIX affinity, thereby implementing the same abstractions present in Linux. The lightweight OSs are further not optimized for database workloads, which pose high latency and scalability constraints on memory management.

This thesis proposes an approach for co-designing the virtual memory subsystem (VMS) of unikernels with database systems to allow better flexibility and control over memory management on the application layer. We introduce the kernel API to integrate kernel and DBMS, thereby providing fast and scalable access to memory management primitives. To that end, we revisit the virtual memory stack of legacy unikernels and propose a new VMS architecture designed to withstand highly parallel database workloads. We implement our design in the unikernel OSv to evaluate its performance using micro and macro benchmarks. Our evaluation shows a five-fold performance boost for database workloads compared to the original OSv architecture.

Contents

1 Introduction

For decades, database systems have run on general-purpose OSs. Although a mismatch between the abstraction provided by the OS and the requirements of DBMSs was recognized nearly half a century ago [15], shipping a custom OS to clients lacked feasibility [19]. Now, the widespread movement toward cloud environments unlocks new possibilities for co-designing DBMS and OS.

Moreover, due to the increased core counts over the last two decades [7], providing competitive performance in the world of databases places stronger scalability demands on OS and DBMS.

Motivation. Recent work [8, 33] evaluates the scalability aspect of the general-purpose OS Linux, highlighting its unsatisfactory performance in regard to memory-mapped I/O (mmio). As a consequence, complex kernel modules [8] or new virtual memory architectures [2, 29, 30] have been proposed.

Furthermore, relying solely on the POSIX application programming interface (API) does not fit the needs of databases. A prominent example is employing hardware-accelerated buffer management [4, 18, 28, 14]. While mmio promises high levels of performance due to translation lookaside buffers (TLBs) and hardware-assisted page table walks, OS policies on page eviction cause correctness issues for databases using ARIES style write-aheah logging (WAL) [19]. As the OS further limits the application's control over page replacement strategies, working sets that don't fit in main memory tend to cause performance degradation beyond the efficiency of a traditional buffer cache approach [4]. This example demonstrates how hiding system-level functionality can cause performance overhead. For the DBMS to utilize hardware acceleration, the OS must provide control over kernel functionality on a lower level.

Address space and privilege isolation are important concepts for general-purpose systems that multiplex multiple, possibly data-sensitive, applications. However, cloud environments often run a single application inside a virtual machine. This strategy mitigates the importance of these principles, as strong isolation is provided by the hypervisor [35]. In the cloud, virtualized applications continue to pay the price for address space and privilege isolation, as system calls still incur the overhead of kernel traps, while its benefits have mostly been mitigated.

Unikernels. The presented drawbacks motivate re-orientation from Linux towards customized kernels for high-performance applications. We propose *legacy* unikernels

[24, 35] as lightweight OS for database applications, as their architecture offers the following benefits. (a) Operating on a single address space, unikernels do not require context switches, thereby avoiding the overhead for kernel-level instructions. (b) While legacy unikernels implement parts of the POSIX API to allow running unmodified Linux programs [26, 37], they enable the implementation of novel interfaces and application-kernel communication. (c) As unikernels focus on simplicity, they run on a small, comprehensive code base compared to general-purpose OSs. This makes them easier to modify and debug, a property vital for a comprehensive co-design of unikernels and database systems.

Research Gap. The original unikernel idea [25] is to provide the application with a language-specific, specialized OS (*clean slate* unikernels). General purpose (*legacy*) unikernels [26, 37], on the other hand, have focused on providing a POSIX confirm environment to maximize the number of applications that run without modifications instead of optimizing the kernel for a single application or type of application. Leis and Dietrich [19] envision exactly this by proposing unikernel-based cloud database systems and present high-level ideas for different OS components. However, the authors do not provide explicit ideas on the design of a unikernel's VMS.

Problem Statement. Currently, legacy unikernels implement the POSIX API to be compatible with a wide range of Linux applications. Providing the same abstractions to the DBMS incurs the same mismatch of kernel and DBMS we observe on Linux. The lightweight OSs are further not optimized for database workloads, which pose high latency and scalability constraints on memory management.

Overview. This thesis explores the opportunities of providing an efficient and scalable unikernel DBMS co-design. To that end, we design a kernel API and expose it to the application level, effectively giving full control over the VMS to the DBMS. For the provided interfaces to be performant, we revisit the virtual memory (VM) stack of unikernels to identify and eradicate bottlenecks. By replacing the physical memory allocator and applying a decentralized design to the VMS, we focus on improving scalability and performance while also considering memory consumption and boot time. To show the feasibility of our design, we implement our system in the unikernel OSv [26] and evaluate our implementation in terms of (1) boot time, (2) memory footprint, (3) synthetic performance and scalability benchmarks and (4) online analytical processing (OLAP) database workloads.

2 Background

This chapter identifies and explains background knowledge relevant to our work.

2.1 Virtual Memory Stack

We provide a concise overview of fundamental memory management concepts in general-purpose OSs. Details deemed insignificant to understand our work are omitted.

2.1.1 Paging

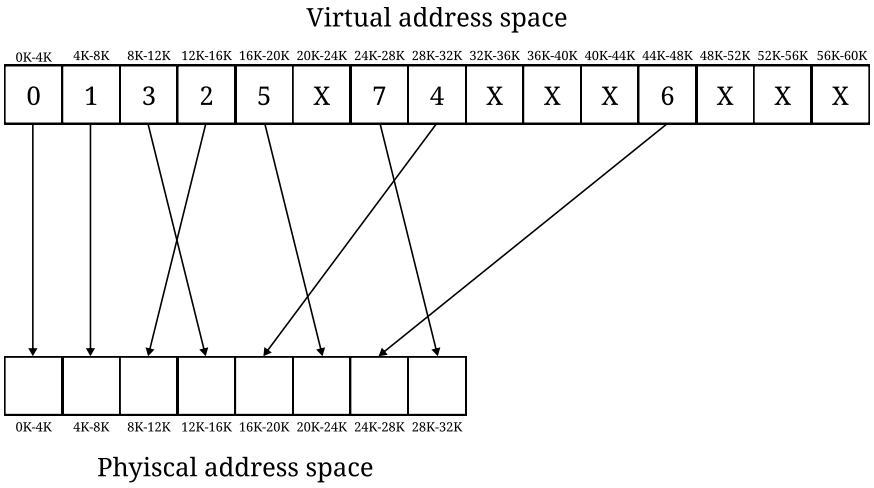


Figure 2.1: Virtual to physical memory mapping. Squares represent a page or page frame. Pages and page frames begin at a multiple of 4 KiB and end one short of the next multiple. For instance, 4K-8K represents the address region 4096-8191. Inspired by Tanenbaum and Bos [36].

Most modern operating systems use the virtual memory abstraction called *paging* [36]. Employing this technique, the address space of a process is divided into fixed-sized pages of virtual memory. A common size for one page is 4 KiB. Each virtual memory page corresponds to a physical page frame of the same size. A physical page is accessed

by its *page frame number*. The memory management unit (MMU) translates virtual to physical addresses before they go on the memory bus. This abstraction has, amongst others, the following implication: There is no correlation between the order of virtual and physical pages. Explicitly, as shown in Figure 2.1, this means that two or more contiguous pages aren't necessarily contiguous in physical memory.

2.1.2 Page Table

The data structure storing the mapping between physical page frames and virtual pages is called *page table*. It efficiently translates virtual to physical addresses [36], with hardware-assisted page table walks on newer central processing units (CPUs).

With some new processors supporting up to 57 bits for the virtual address space [36], a flat page table implementation (hash table) would be too large to fit into random access memory (RAM), even on large machines. Instead, modern operating systems implement *multi-level paging*, where the page table follows a tree structure with up to five levels [36]. This reduces memory overhead by dynamically allocating parts of the page table when a mapping within the requested region is made. Furthermore, mappings that span over multiple lower page table levels can be compacted and stored with less memory overhead further up the hierarchy.

2.1.3 Translation Lookaside Buffer

Multi-level paging presents additional memory accesses per address translation, as more page table levels must be read. This is known as *page table walk*. To mitigate this overhead, modern processors employ TLBs, which cache recently used page table entries to accelerate address translation. As TLB lookups are usually equivalent to L1 cache lookups on modern CPUs, they are orders of magnitude faster than a lookup in the page table hierarchy [36].

2.1.4 File Mappings

By utilizing the `mmap` system call, Linux allows mapping files that reside on secondary storage directly into the virtual address space. Once a virtual address within the mapping is accessed, the kernel loads a part of the backed file into a *page cache* in main memory. Read and write operations are executed on the cached copy, and synchronization with secondary storage is left to the operating system but can be explicitly triggered using the `msync` system call.

2.1.5 Page Eviction

As depicted in Figure 2.1, paging allows the virtual address space to exceed physically addressable memory. This enables using more memory than physically available by employing *page replacement* [36]. Page replacement allows the operating system to write back (*evict*) page frames to secondary storage to free up main memory, thereby creating the illusion of additional physical memory. Addressing a page currently written to secondary storage will then cause a *page fault*, ultimately triggering the page to be brought back into main memory.

While this allows applications to exceed the limitations of physical memory, writing and reading pages from secondary storage incurs significant performance penalties. Therefore, the kernel employs a *page replacement algorithm* aiming to minimize this overhead by determining which pages to evict [36]. Even though system calls like `mlock` and `msync` give applications some control over locking pages in main memory and when to synchronize modified (*dirty*) pages with potential copies in non-volatile memory, the application cannot prevent the operating system writing back dirty pages.

2.1.6 Virtual Memory Areas

Once created, page table entries don't show individual memory mappings, neighboring anonymous mappings could have been created with any number `mmap` calls. To view VM mappings individually, the Linux kernel maintains, in addition to the page table, a set of virtual memory areas (VMAs) to store memory mappings [22].

Every mapping the user requests is guaranteed to be inserted into the VMA set, while the page table only stores currently active mappings [4]. Therefore, looking up page faults is done by searching the kernels VMA set for the corresponding entry, followed by populating the page table with the necessary data.

2.1.7 Heap

General-purpose operating systems rely on address space isolation to protect the data owned by each process. As such, they require a way to allocate memory in their own address space. A naive approach would be to call `mmap` for every allocation. However, system calls require a kernel trap, making them expensive even without considering contention and the increasing number of VMAs in the system. To mitigate this, general-purpose operating systems use a memory abstraction that buffers a larger region of memory. As the process already owns the entirety of the buffer, user-level privilege is sufficient to allocate memory from it, effectively reducing the number of system calls and VMAs in the system. This abstraction is commonly known as the heap.

Besides buffering a larger memory area in the process's address space, the heap serves a second purpose: Splitting pages into suitable sizes for small memory allocations as the page frame allocator cannot serve requests smaller than a single page.

2.2 Databases

2.2.1 Buffer Management

Traditionally, databases utilize a buffer pool to compensate for insufficient main memory. This pool allows the majority of memory to be stored on cheap secondary storage while relevant data is kept in main memory [21]. The page movement between primary and secondary memory is usually implemented utilizing `read` and `write` system calls, therefore maintaining control at the DBMS level [4].

2.2.2 In-memory Databases

In the last 30 years, main memory prices have dropped significantly [21], increasing the feasibility of databases residing entirely in RAM. So-called *in-memory* databases exhibit less overhead, as no file I/O or expensive buffer cache operations are necessary [9]. At the same time, pure in-memory database systems do not support data sets exceeding the available main memory [14].

Due to the improvement of secondary storage's throughput and latency in the last few years [39], the concept of keeping all data in volatile storage has lost significance. The superior NVMe performance/price ratio instead moves the focus back to DBMSs that are capable of storing data sets larger than main memory on non-volatile storage.

2.2.3 State of the Art

To combine the benefits of buffer management with in-memory databases, novel approaches include hardware-assisted page replacement by relying on virtually mapped memory, often utilizing the `mmap` system call. The virtually mapped memory allows the operating system to transparently evict pages to secondary storage without incurring additional overhead caused by system calls [4]. Furthermore utilizing the TLB and hardware-assisted *page table walks* seems to make `mmap` a promising solution to enable support for data sets exceeding main memory. While Crotty et al. [4] show that using file mappings instead of a buffer manager exhibits worse performance while incurring safety concerns, modifying the Linux kernel [18, 28] produces promising results for virtual-memory-assisted buffer management.

2.3 Unikernels

Unikernels are specialized, lightweight operating systems designed to host a single application [24]. The application is statically linked to the kernel image at compile time, resulting in a single-purpose OS tailored to cloud environments.

The library OSs are categorized into (a) clean slate [25] or (b) legacy [26, 37]. The first is often language-specific and does not try to emulate traditional OSs in any way [35], while the latter implements a subset of POSIX functionality to run unmodified Linux executables. Both classes share the limitation of a single address space, restricting applications to a single process.

By relying on the hypervisor for isolation, the unikernel design safely avoids address space and privilege isolation [35]. This reduces the complexity of the overall system and increases performance, as system calls no longer suffer from overhead induced by kernel traps. Due to their simplicity, unikernels provide a narrowed attack surface [35], resulting in acceptable security.

3 Overview

We begin by outlining the promising potential of Unikernel/DBMS co-design for the VMS and itemize the key design objectives to accomplish efficient and scalable kernel integration. Subsequently, we overview the resulting system architecture and discuss prominent design challenges.

3.1 Potential of Unikernel/DBMS Co-design

As shown in Chapter 1, relying on traditional mmio interfaces provided by general-purpose OSs does not offer the desired performance nor the correctness guarantees required by most database systems. DBMSs therefore often implement custom buffer managers to bypass mmio and gain the required flexibility and control over page caching and page replacement strategies. In addition to losing hardware acceleration, buffer cache utilization forces a trade-off between complexity and performance [14, 18].

An approach to avoid this complexity inside the DBMS design is the implementation of kernel modules [18, 28] tailored to the needs of a specific database, thereby making mmio feasible. While this mitigates initial correctness concerns and can boost performance, the DBMS loses flexibility and control compared to the buffer-management approach, as only limited kernel functionality can safely be exposed employing this method. Furthermore, Linux kernel modifications are not trivial due to the complexity of Linux's VMS and require maintenance due to progressing kernel versions, ultimately resulting in high complexity for the overall system.

Both approaches suffer from a static trade-off between complexity, flexibility, and performance. We argue that the unikernel/DBMS co-design enables a more dynamic trade-off and better overall performance perspectives. Due to the single address space, kernel modifications are not limited by privilege isolation, allowing full flexibility and control to be exposed to the application level without incurring security risks. Contrary to current techniques, this facilitates the combination of designated components of the system-level mmio implementation tailored to databases with custom functionality optimized for a specific database. Our approach enables DBMSs to fully exploit hardware acceleration and build on top of existing, optimized infrastructure while gaining an additional performance boost due to context switch elimination.

3.2 Design Goals

Flexibility. Due to the limited flexibility of mmio (i.e. `mmap`) in Linux, DBMSs implement kernel modules to gain the desired functionality [18, 28]. To ensure kernel modifications of any kind are unnecessary in our design, we must provide optimal flexibility to the DBMS. The kernel API must supply complete functionality on a low level to allow generic use cases. We aim to provide a system that enables the application to re-implement single components of the VMS without compromising the functionality of neighboring system-level features.

Scalability. In the last two decades, the number of cores in a system increased significantly [7], raising the performance impact of scalability. Unlike Linux, by redesigning the VMS for a unikernel, we are not bound by the complexity of a huge code base and entanglement of system components [22]. This gives us the opportunity to specialize the design of our system explicitly for highly parallel database workloads without incurring overhead due to trade-offs made in favor of compatibility and generality. We aim to harness multicore performance throughout the virtual memory stack by optimizing maintained POSIX interfaces and providing excellent scalability for the introduced kernel API.

Latency. For database applications, it is essential that single-memory operations delegated to the underlying operating system execute quickly and scale well in terms of increasing main memory and number of VMAs in the system. Choosing scalable algorithms and data structures is vital to ensure low latency and high throughput for all database-related memory operations as well as introduced interfaces for the OS/DBMS co-design.

Memory Efficiency. Fitting larger parts of a data set into RAM reduces the number of page faults, thereby increasing overall latency and performance. We aim to reduce the system's memory footprint to optimize the utilization of limited main memory in a unikernel environment. By reducing the overhead associated with bookkeeping and virtual memory management, we aspire to provide a lightweight operating system for cloud databases.

Compatibility. Most of today's database systems are designed to run on general-purpose OSs, particularly Linux. Therefore, maintaining POSIX compatibility for the most common system calls allows running unmodified databases. Our VMS design must preserve compatibility with traditional Linux functionality while moving past the scalability and flexibility limitations of the general-purpose kernel.

3.3 System Overview

In this section, we present a high-level overview of the system architecture. We iterate over unikernel components that are subject of our work and motivate changes made to the system. Low-level details have been omitted in favor of clarity.

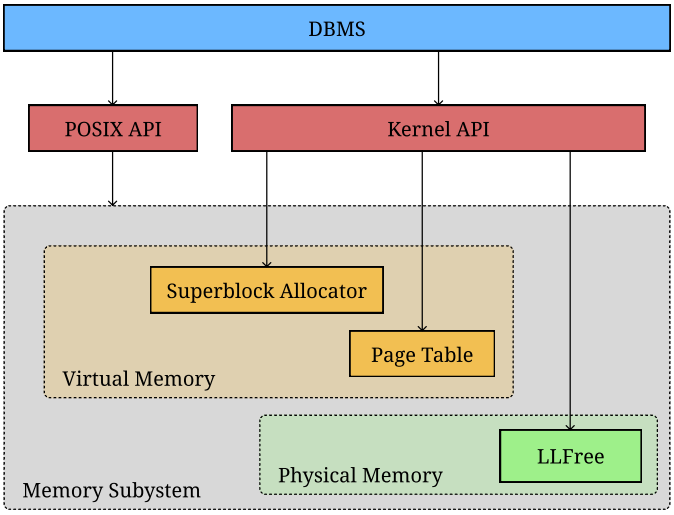


Figure 3.1: High-level view of the memory subsystem and the interfaces between DBMS and unikernel.

Figure 3.1 depicts a high-level view of the memory subsystem and its integration with the DBMS. The POSIX API and page table are largely left unmodified, limiting the modifications by our design to the following components:

Physical Memory Subsystem. We employ LLFree [38] as page frame allocator, allowing fast and concurrent allocation of physical memory. The allocator's design furthermore enables physically contiguous memory allocation of huge pages.

Virtual Memory Subsystem. We introduce the *superblock allocator*, a virtual memory architecture focusing on decentralization and core-locality. Its general design can be applied to existing virtual memory systems while keeping most interfaces unchanged.

Kernel API. To grant the DBMS access to the internals of the memory subsystem, we expose a specialized kernel API to view and modify the data structures of the VMS, including the page table, in a safe and efficient way. The API also allows for the direct allocation of physical memory and can be used to gather extensive information about the system's memory state.

3.4 Design Challenges

Kernel API Intricacy. Optimal flexibility incurs complexity by demanding low-level control of virtual memory functionality. At the same time, exposing low-level control may collide with thread safety as some operations must be seen in a broader context. We overcome this challenge by organizing API endpoints by component. Each component, like the VMA storage or virtual memory allocator, is the smallest unit inside the VMS without implementation-specific dependencies to others. Each component can be utilized when overriding VMS functionality in user space or left out of the custom implementation. Synchronization introduces a second dimension into the VMS separation: By address range. Interfaces provided by the kernel API are not thread-safe by design but require the lock for the corresponding component and address range.

Adaptability. Adapting a DBMS to the kernel API specified in this work requires changes in the DBMS code base. This poses an immense challenge for complex database systems, making the utilization of kernel interfaces unrealistic. While this issue cannot be entirely alleviated for a general API design, we propose *POSIX API interception*. By giving the DBMS control over VMS related system calls, fewer interfaces must be adopted, leaving most of the DBMS code base unmodified.

Heap Exigency. General purpose OSs rely on a heap to minimize system call overhead, as detailed in Chapter 2. This allows a simple correlation where the available memory for a process equals the free memory on the heap. For unikernels, on the other hand, the library OS design alleviates overhead connected to OS primitives, making user space memory buffers redundant. Additionally, memory available to the single application equals the entire system's free memory rather than the heap capacity. Therefore, buffering memory on the heap or in other indirections provides limited benefits while distorting overall memory metrics and introducing fragmentation. At the same time, we recognize the performance derivation between different heap allocators for database workloads [6]. Our design relinquishes dictating a specific heap allocator by shrinking the heap to its core functionality. The system design thereby maintains simplicity while the unikernel's POSIX affinity allows seamless integration of custom memory allocators on the application level.

Initialization Complexity. For cloud environments, where instances are often spun up and down to handle the varying workloads, boot time is crucial. Unikernels achieve this by minimizing the initialization complexity of their subsystems. Our design, however, incorporates more intricate components, adding complexity to the boot process. By reordering the initialization procedure, we are able to utilize the superior performance of our design during the remaining boot stages, ultimately mitigating time loss incurred by the complexity increase of the memory subsystem.

4 Design

This chapter illustrates the proposed kernel API and the design of the revised unikernel components to achieve our goals stated in Chapter 3 and elaborates on changes made to the unikernel OSv to support the new architecture.

4.1 Kernel API

To ensure isolation between different address spaces, Linux hides most of the memory management information from the user level. Previous work shows that the provided abstractions do not satisfy the required flexibility for a DBMS [4, 18, 20, 14]. The single address space in unikernels mitigates the security concerns limiting Linux functionality and allows our design to greatly extend the functionality exposed to the application. This section outlines the key ideas of our design, enumerates the endpoints provided to the DBMS, and details synchronization primitives.

4.1.1 Physical Memory Access

The DBMS can benefit from knowing the amount of currently free physical memory. This is especially relevant to prevent out-of-memory (OOM) errors for unikernels that do not implement page replacement. Furthermore, the application may require physical memory allocation to populate the page table. As detailed in Table 4.1, our API enables the application level to allocate physical memory of a given order and request statistics regarding physical memory utilization. Since page frames cannot be accessed by their physical address, the kernel API returns the page frame's linearly mapped representation in virtual address space. Due to the thread safety of LLFree and the simplicity of physical-to-virtual address translation, no synchronization is required for any requests solely concerning physical memory.

4.1.2 Page Table Access

Due to missing privilege isolation, the application can read the `cr3` register to get the root element of the page table. However, this does not suffice to safely modify the kernel data structure. Additionally, we expose the means to synchronize page table

Interface	Description
`void* frames_alloc(order)`	Allocate physically contiguous memory of the specified order and return the linearly mapped address.
`void frames_free(addr, order)`	Free the physically contiguous memory represented by the linearly mapped address.
`u64 stat_free_phys_mem()`	Get the amount of free physical memory.
`u64 stat_total_phys_mem()`	Get the amount of total physical memory.
`void flush_tlb_all()`	Flushes all TLBs in the system.
`mutex& pt_high_mutex()`	Returns the mutex for higher page table levels for modifications of linear mappings.
`rwlock_t& vma_lock(addr)`	Returns the read-write lock for an address range containing the VMA intersecting with the specified address.
`rwlock_t& free_ranges_lock(addr)`	Returns the read-write lock for the memory range set containing the address.
`void insert(vma)`	Insert the VMA into the kernel's internal state.
`void erase(vma)`	Erase the VMA from the kernel's internal state.
`optional<vma*> find_intersecting_vma(addr)`	Find the VMA intersecting with addr, if one exists.
`vector<vma*> find_intersecting_vmas(addr, size)`	Find all VMAs intersecting with the address range specified.
`bool validate(addr, size)`	Test if the given region can be allocated.
`void allocate_range(addr, size)`	Allocate a given virtual memory region.
`uintptr_t reserve_range(size)`	Allocate any virtual memory range of the given size.
`void free_range(addr, size)`	Free the given virtual memory range by returning it back to OSv's free lists.

Table 4.1: Kernel API table. VMA operations and `allocate_range` are not thread-safe and require synchronization utilizing the `vma_lock` and `free_ranges_lock` respectively. POSIX API interception is not included as the specification is identical to existing system calls.

modifications between kernel and DBMS by providing a lock for the high page table layers and selective locking of specific areas for the lower levels (see Table 4.1). In combination with physical memory allocation capabilities and an endpoint to flush the TLB, the application can create its own page table entries and modify existing ones.

4.1.3 VM Access

While page table access is sufficient to create and alter page table entries, this merely modifies the mirror of a temporary system state. To create lasting mappings in accordance with OS policies, the DBMS requires access to the system's virtual memory allocation and VMA storage functionality. This includes endpoints to reserve predefined virtual memory regions and allocate arbitrary memory regions of the given size. To improve flexibility, VM allocation must be distinct from VMA creation. Therefore the functionalities constitute different endpoints in the interfaces specified in Table 4.1.

Furthermore, the system may create a new memory mapping without updating the page table. Instead, a VMA entry is created and stored in internal data structures. To enable the application layer to view and modify VMAs created by the system, our API exposes the functionality to find all VMAs in a specific virtual memory region as well as insert and delete VMAs from system data structures.

4.1.4 Synchronization

The design of our VMS requires synchronization on distinct subsets of virtual memory regions. Our API exposes references to the locks used by internal data structures to guarantee sequentialized access to critical sections. The precise locking behavior of internal data structures is explained in Section 4.3.

4.1.5 POSIX API Interception

Overwriting system functionality without modifying the kernel code requires an abstraction layer above the system call implementation that enables intercepting and propagating requests to user-defined functions. We propose an interception layer between DBMS and unikernel to catch system calls related to VM management. Thereby minimizing compatibility efforts on the DBMS side by providing a gateway to modify the entire VMS by overriding system call functionality. As we expose optimized, self-contained components with the kernel API, they can be easily integrated into new VMS designs. Meanwhile, our design hides the complicated and entangled implementation-specific functionality of unikernels for which no general interfaces can be designed.

4.2 Physical Memory Subsystem

The design of the page frame allocator we integrated in OSv is split into two parts: the early allocator and the LLFree allocator.

4.2.1 Early Allocation

The early allocator is used during the early boot stages. Its primary objective is to allocate larger memory regions for an initial linear mapping. Since memory can only be accessed by its virtual, not physical, address, this step is required to access any physical memory provided by the hypervisor. Secondly, the early allocator is crucial to allocate the internal data structures of LLFree.

Early memory allocation focuses on simplicity and a low memory footprint. As the early allocator becomes obsolete before symmetric multi-processing (SMP) is enabled, contention and cache sharing do not need to be considered for its design. Furthermore, the allocator handles a constant number of allocations during boot, making potential performance drawbacks and scalability concerns negligible.

Based on these factors, early memory allocation is achieved by maintaining a single linked list of free physical memory. The information for each node is stored *in-place* at the beginning of the physical memory region it represents.

4.2.2 LLFree

We propose LLFree [38] as a lock- and log-free allocator. It is designed to be scalable and cache-friendly while having a low memory footprint. Additionally, it is capable of allocating contiguous memory in 11 different orders. Dependent on the page frame size P which is defined in the MMU, the size of order n corresponds to:

$$2^n \times P \text{ with } n \in [0, 10]$$

Limiting the different allocation sizes to orders instead of arbitrary multiples of page size enables anti-fragmenting behavior while supporting the allocation of contiguous memory up to a huge page (order 9). This is vital for fast allocations of large memory areas, as it can greatly improve the efficiency and memory overhead of populating the page table.

4.3 Virtual Memory Subsystem

4.3.1 Memory Mapping

Linear Mapping. By only having to maintain a single address space, some unikernels avoid the overhead of virtual memory and page table population for most of their memory allocation requests. By linearly mapping the entire physical memory into the single address space, the virtual address of a page can be based solely on a logical translation of the physical address. This allows for fast allocation as only the physical memory allocator needs to be invoked. Therefore avoiding the performance overhead of virtual memory allocation and the connected memory overhead caused by page table population, linearly mapped memory allocation shows exceptional performance for a majority of use cases, especially if the overall memory usage is constrained.

The linearly mapped memory allocation approach faces a trade-off between fragmentation and high memory overhead, making it unsuitable for applications with high memory consumption. Fragmentation arises because the allocator must support a wide range of possible allocation sizes, which can lead to an accumulation of small chunks of memory. On the other hand, limiting the range of physical memory sizes to reduce fragmentation causes allocated regions to greatly exceed the requested size, resulting in higher memory overhead. Linearly mapped memory allocation further disallows lazy allocation which can be used to reduce overall memory consumption.

Non-Linear Mapping. We argue that utilizing virtual memory concepts for all allocations provides benefits that ultimately outweigh the additional overhead caused by virtual memory allocation and page table population:

The memory does not need to be contiguous in the physical address space. The page table can be populated with arbitrary page frames, adding up to the exact number of pages requested. Thereby, precisely the required amount of physical memory can be allocated, regardless of the limitation imposed by LLFree to only allocate in orders. This reduces fragmentation in physical memory, as page sizes allocated by LLFree are consistent.

Additionally, keeping a VMA for every allocation makes tracking the size of physical allocations in the form of a header obsolete. As our design avoids splitting or merging VMAs, the order of each page allocated by LLFree can be derived from the VMA size upon deallocation. While this would save a maximum of one page for OSv's page range allocator, LLFree can suffer from a huge memory overhead where nearly double the amount of requested physical memory must be allocated to store the header.

4.3.2 Virtual Memory Allocation

Fine-grained mapping of memory regions offers excellent flexibility and efficient memory utilization. However, it also increases the number of VMAs significantly. In addition, allocations suffer from the overhead of relying on physical and virtual memory allocation. Consequently, optimizing virtual memory allocation is essential to reduce this overhead and lay a foundation for efficient virtual memory management.

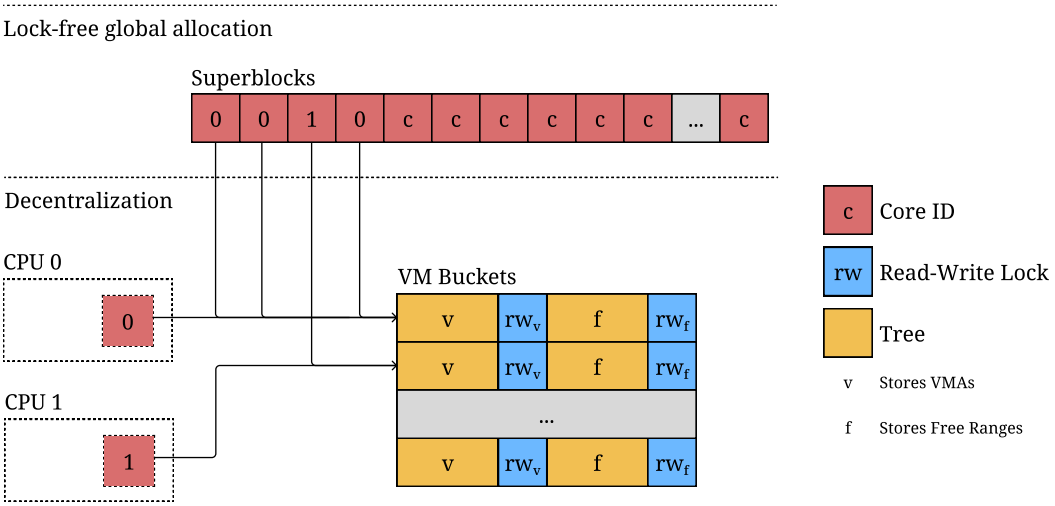


Figure 4.1: The architecture of the virtual memory subsystem. Superblocks and VM Buckets are each stored in an array. Dotted lines imply that the information is not stored in memory directly but is derived dynamically.

Figure 4.1 depicts the architecture chosen to reduce contention and enhance allocation time for virtual memory regions. We propose a decentralized design focusing on scalability and cache locality. Our design modifies OSv's virtual memory subsystem in two key aspects:

Decentralization. Instead of relying on a single global allocator, each core manages its own free virtual memory ranges. Furthermore, each core maintains the VMAs within the bounds of this memory, thereby improving locality and reducing synchronization overhead. In our decentralized design, each core allocates larger chunks of virtual memory (*superblocks*) from a global pool. The size of every superblock is a pre-defined multiple of the page size and is consistent throughout the system. These superblocks are added to the cores *free memory ranges* set from which individual requests are served.

Even though tracking free memory ranges instead of allocated ones does not lead to a direct performance benefit, it offers better modularity, by allowing to consider virtual memory as a set of isolated superblocks rather than a single contiguous address space.

This approach enables the distribution of superblocks across cores without additional bookkeeping overhead.

Lock-free global allocation. The VMS maintains a global array of superblocks, where each array index can be translated to the beginning of the superblock's virtual memory region. Since all superblocks are of equal size, this translation does not require additional metadata. Each array entry stores an atomic state value, indicating whether the superblock is free or reserved. The size of the atomic value must be chosen to support more non-free states than cores in the system to provide sufficient space to store the reserving core's ID. At allocation, the array entry for the corresponding superblock is atomically compared and swapped with the calling core's ID. This results in a superblock array that can be employed as a hash table to find the ID of the core that reserved the superblock containing a given address in constant time.

4.4 Algorithms

This section sketches the control flow of the three major operations on virtual memory. Some edge cases have been omitted in favor of readability.

4.4.1 Mapping Virtual Memory

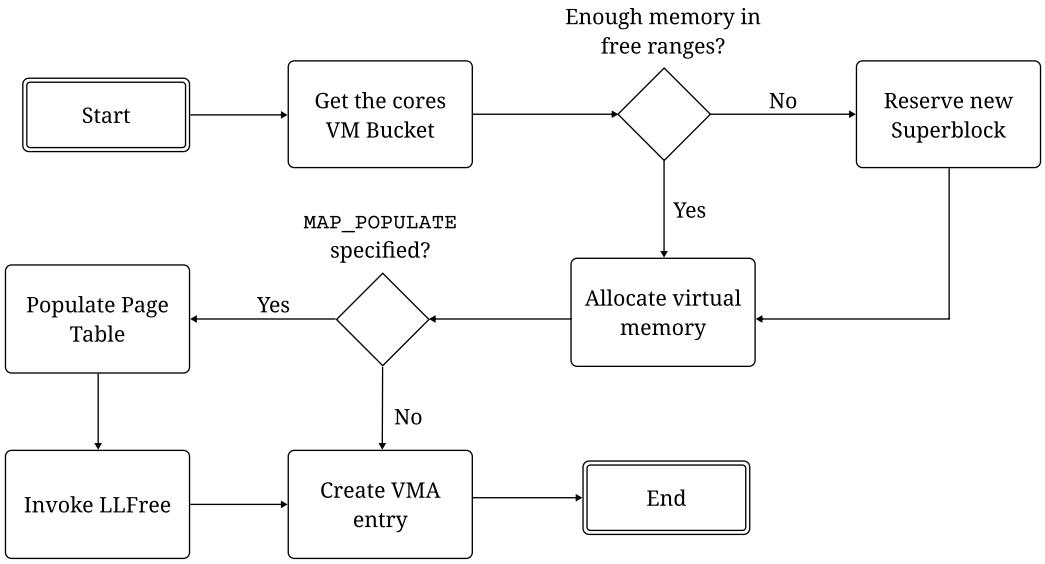


Figure 4.2: The flow of creating an anonymous memory mapping and potentially backing it with physical pages.

Every core has a designated VM Bucket. For mapping requests with `MAP_FIXED` not set, virtual memory regions to be chosen for allocation are drawn from this bucket. If the bucket does not contain a large enough memory range, a new superblock is allocated and inserted into the bucket. The page table is populated depending on the `MAP_POPULATE` flag. During this process, LLFree can be invoked several times to back the virtual memory with page frames. Lastly, the VMA entry is written to the bucket's VMA set. Figure 4.2 sketches this process, including page table population.

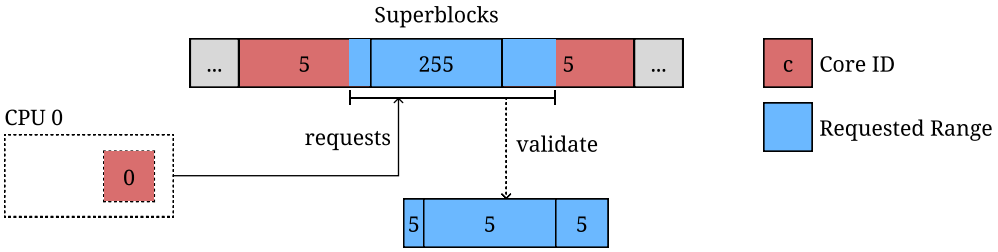


Figure 4.3: Example of owner validation during a `MAP_FIXED` memory mapping.

The requirements `MAP_FIXED` poses on the virtual memory subsystem increase the complexity of the allocation process significantly. As there is no guarantee the requested region resides within one superblock, such requests must be validated to ensure no ownership conflicts of involved superblocks. During validation, one of the following scenarios may occur:

- **All Free**: If all involved superblocks are available for allocation, they are acquired by the calling core and the request proceeds.
- **One Owner**: If a subset of superblocks is owned by core A, the calling core allocates the remaining superblocks in A's name and proceeds with the request inside the corresponding bucket. Figure 4.3 exemplifies this scenario. Should one of the superblocks be temporarily reserved for an optimistic allocation of multiple superblocks, the calling core waits for the outcome of the allocation as a failure during optimistic allocation could still result in the superblock being released.
- **Multiple Owners**: The request cannot be fulfilled. While allocating memory ranges from different buckets does not pose a problem, allowing cross-bucket VMAs invalidates hash table lookup. Migrating all superblocks in question poses an even higher overhead. Therefore, our design accepts this slight derivation from POSIX-specified functionality.

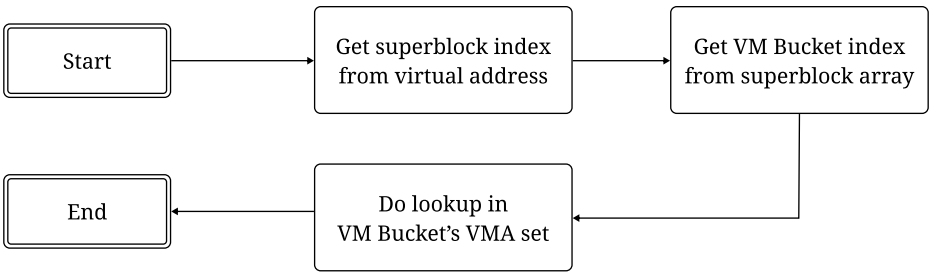


Figure 4.4: Lookup process for virtual memory areas.

4.4.2 Lookup

Figure 4.4 shows the high-level logic to lookup VMAs. The superblock index is calculated using a static shift and offset addition from the faulting address. Afterward, the superblock array is utilized as a hash table to determine the VM bucket that contains the corresponding superblock.

4.4.3 Unmapping Virtual Memory

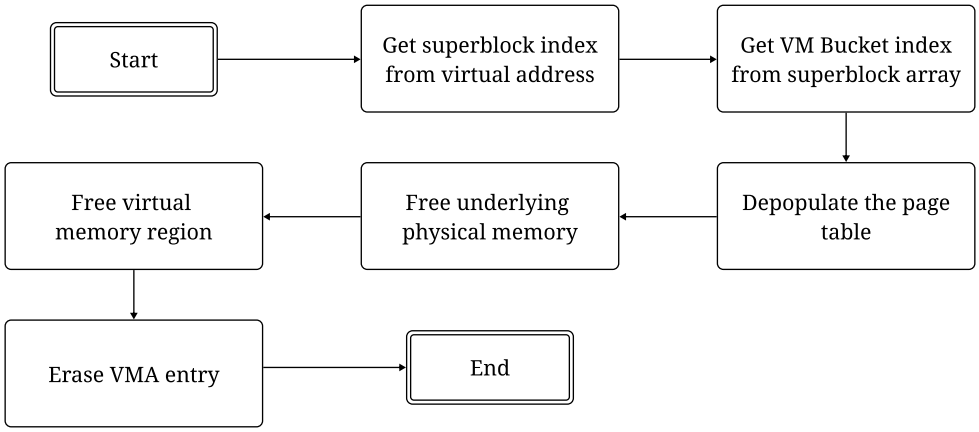


Figure 4.5: Control flow of unmapping a memory region.

Figure 4.5 depicts the process to unmap a VMA. The corresponding superblock index is calculated from the virtual address, followed by a hash table lookup to determine the VM bucket containing the VMA. The page table is depopulated, potentially freeing mapped page frames, and the free memory range is returned to the bucket. Potentially existing neighboring ranges already residing in the bucket are merged with the new entry. Lastly, the VMA entry is removed from the bucket's VMA set.

4.5 Heap Allocator

Our design avoids the integration of an elaborate heap allocator on the system level to prevent unnecessary complexity and increased boot times. Especially for database applications, which often implement special memory abstractions in the form of arena allocators or utilize custom memory allocators [34, 12, 5, 17], it is not essential to provide an optimized unikernel-native implementation. However, the system functionality requires `malloc` and related endpoints to provide the expected functionality. Therefore, our design employs the following strategies to satisfy incoming requests.

Small Allocations. The heap allocator must handle allocations smaller than page size, as the physical memory allocator cannot directly fulfill those. Our design utilizes the mechanism already implemented in the unikernel. For OSv, this functionality is called the *memory pools*. The unikernel occupies 13 memory pools to serve requests of different sizes. Each pool consists of a 4 KiB page that is split into equal-sized chunks. The pool with the smallest chunk size, large enough to satisfy the request, is selected, and one of its elements is returned. Once every chunk in a pool is allocated, the pool requests a new page [26]. As the buffer sizes of this method are already minimal, it fulfills the requirements imposed by our design goals.

Large Allocations. When relying on linear mapping, the physical memory allocator usually directly satisfies allocation sizes from page size upwards. Instead, we utilize the optimized VMS to avoid drawbacks connected to this approach and create a new anonymous memory mapping for every request.

5 Implementation

In this chapter, we describe the implementation of our memory subsystem. We outline changes made to OSv, explain which components we disabled to fit our design, and present implementation-level details of new components.

5.1 Page Frame Allocation

We embed the c-version of LLFree [23] into the unikernel. Initialization of LLFree requires memory allocation for several data structures. Therefore, we implement an early memory allocator that can be initialized without memory allocations.

5.1.1 Early Memory Allocation

Data Structure. When the physical memory is set up, *free memory ranges* are stored in an `boost::intrusive` list. Unlike the implementation from the standard library, this makes the caller responsible for allocating individual nodes. We utilize this property to store each node at the beginning of the free memory range. Hence, the node's physical address is simultaneously the beginning of the free region. This avoids allocating memory for the node and reduces the size of the node by 8 bytes on 64-bit architectures, as the address of the memory range does not need to be stored separately.

Granularity. While this design is capable of allocating with byte precision, all requests are aligned to page size, allowing the pages to be freed with either LLFree or the early allocator. This further allows utilizing OSv's linear mapping scheme. OSv distinguishes between allocation categories: (1) smaller page size, (2) exactly page size and (3) larger one page. For each category, the physical address is mapped to a specific virtual memory region. When the memory is freed, a mask can be used to determine its category. Therefore, no header is required for single pages as the size of the memory region is already known to the system.

Algorithm. Booting OSv on x64, the early allocator handles 19 requests. We, therefore, neglect fragmentation and performance in the implementation, focusing on simplicity. Requests are satisfied with the first memory region large enough to provide the requested size. Furthermore, all requests are serialized on a global mutex, as the early boot mechanism becomes obsolete before SMP is set up.

5.1.2 LLFree initialization

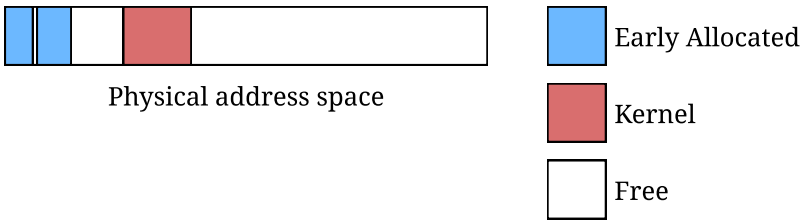


Figure 5.1: Example of physical memory layout during LLFree initialization.

Physical Memory Size. After all memory regions are added to the early allocator's free list, the highest physical memory address can be determined. As LLFree is designed to work on contiguous memory [23], this address is required for its initialization. By handling gaps in physical memory later, the page frame index returned by LLFree can be translated into the corresponding physical address with an efficient shift operation rather than requiring post-processing on allocation and pre-processing on deallocation for ranges of physical memory that are already allocated or inaccessible. Figure 5.1 shows an exemplary layout of the physical memory during LLFree initialization, including already allocated memory regions.

State Synchronization. On the other hand, when initializing LLFree with the entire span of physical memory, any address, including invalid ones, may be returned by the page frame allocator. We solve this by targeted allocation of unavailable memory after initialization. By iterating over the early allocator's free list, invalid pages are identified and marked as allocated in LLFree using its `llfree_get_at` interface. This method does not significantly increase boot time, as we will show in Chapter 6.

Number of Cores. The setup of physical memory has a high priority during the boot process, while SMP initialization takes place at a later stage. Thus, the number of cores in the system is not known when physical memory is set up. However, LLFree maintains state per core, requiring the number of cores upon initialization. As relying on the early allocator until the number of cores is known nearly doubles the number of allocations, particularly increasing complexity during state synchronization, this approach would significantly increase boot time. Instead, LLFree is initialized with 64 Cores, the highest number supported by OSv. As the metadata required per core by LLFree is 64 Bytes, the memory overhead of initializing the page frame allocator with 64 cores is exactly one page, leaving a maximum of 4032 bytes unused in case OSv is booted with one core. This measure, therefore, does not pose significant memory overhead. Initializing LLFree with more cores than ultimately used does furthermore not reduce performance, as trees are assigned to cores upon first allocation, not initialization.

5.1.3 Obsolete Components in OSv

OSv's original physical memory allocator employs several abstractions to improve performance and scalability as well as reduce the memory footprint.

Page Pools. OSv's original physical memory allocator has a similar, although more optimized, design as the early allocator described above. Particularly it also requires global synchronization. To keep the contention to a minimum, the unikernel utilizes two layers of page buffers. The *L1* and *L2 page pools*. The L1 page pools serve all allocation requests up to page size. Similar to the L1 CPU cache, L1 pools are core-local. They have a pre-defined capacity of 512 pages. Working with thresholds (so-called *watermarks*), the pools trigger a helper thread to *fill* or *unfill*, allowing the actual page allocation to happen concurrently to program execution. For the L1 pool, this means requesting a *batch* of pages from the larger L2 pool. While exhibiting behavior similar to the L1 pools, the L2 pool is shared between all cores and dynamically sets its maximum capacity and watermarks depending on the number of cores. The L2 pool *fills* and *unfills* by requesting pages from the page range allocator [26]. For our implementation, LLFree provides a highly parallel and efficient way to allocate physical pages lock-free. Our implementation, therefore, does not benefit from page pools. On the contrary, memory buffers introduce fragmentation and complexity. Consequently, our implementation does not utilize these buffers.

Reclaimer Threads. A considerable downside of using per-core memory buffers is the incurred fragmentation. A request for physical memory might throw an OOM exception, even though local buffers of other cores contain enough free memory. To solve this issue, OSv relies on tracking the free memory in the system and activating *reclaimer* threads to get memory back from the L1 pools into globally accessible data structures. As our implementation does not employ page pools, the reclaimer functionality is disabled.

5.1.4 Unikernel Integration

LLFree is integrated into existing interfaces of OSv, enabling it to be used with the original memory subsystem as well as the one we present below. The page frame allocator provides interfaces for single-page allocation as well as higher orders. For every allocation, the caller must decide whether a header should be allocated in front of the memory region, tracking the allocation size. When using LLFree with the original implementation, memory regions larger than one page will be mapped linearly in most cases, requiring a header to track the order of allocated memory [26]. Due to LLFree allocating in orders, a header can drastically increase the memory overhead. In case the current order is exceeded, twice the physical memory needs to be allocated.

5.2 Virtual Memory Subsystem

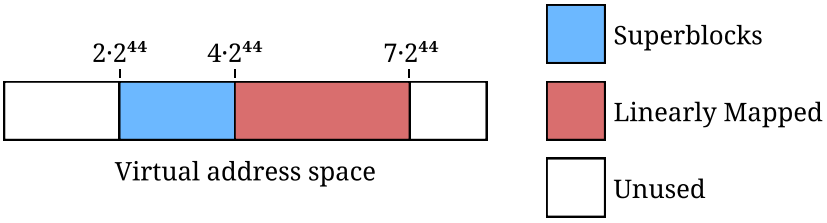


Figure 5.2: Virtual address space layout in OSv. Smaller mappings like the kernel or driver mappings are omitted.

The virtual address space can be split into several distinct parts. Figure 5.2 depicts a coarse overview of the largest sections. Addresses from the unused areas must be specifically requested by the caller and are mostly used for kernel and driver mappings. in detail in this section. The focus of our implementation lies in the superblock range and connected functionalities.

5.2.1 Superblocks

The superblock array contains 32,768 entries, each representing 1 GiB of the superblock region depicted in Figure 5.2. Each entry consists of an atomic byte to track the state of the corresponding superblock. A one-byte state value is sufficient for OSv, as the number of cores is limited to 64 [26]. The superblock array therefore has a total size of 32 KiB in our implementation, as no further information must be stored in the array entries. Array indices are sufficient to identify the corresponding superblock.

```
uintptr_t superblock_ptr(const u64 superblock){
    return (superblock << superblock_bits) + superblock_area_base;
}
```

Figure 5.3: Superblock index to first virtual address within superblock.

As all superblocks are of the same size, translation between the superblock index and virtual memory address can be achieved with an efficient shift and offset addition, as shown in Figure 5.3.

Initialization. Following physical memory initialization, described in Section 5.1, the virtual memory, including the superblock array, is set up. Therefore, all superblocks are set to the value 255, representing a free state.

Allocation. As the superblock array is stored contiguously in memory, the CPU can pre-fetch subsequent entries, resulting in superior linear search performance, compared to linked lists and most tree structures. Our implementation utilizes this runtime optimization to efficiently traverse the array from the beginning when allocating new superblocks. In combination with the first fit algorithm, this method is sufficient to reduce fragmentation, as mostly single superblocks are required. To allocate a single superblock, the calling core uses a compare and swap (CAS) operation, exchanging 255 (`free_idx`) with its core ID.

```
for(unsigned j{i}; j < i+n; ++j){
    if(!superblocks[j].compare_exchange_weak(
        free_idx,
        reserved_idx,
        std::memory_order_acq_rel
    )){
        // Someone else was faster, we have to start over
        release_superblocks(i, j-i);
        return allocate_superblocks(n);
    };
}
```

Figure 5.4: Section of `allocate_superblock` function for optimistic allocation. Variable `i` equals the index of the first free superblock and `n` the number of required blocks.

Although the superblock size should be chosen large enough to satisfy most allocation requests, thereby limiting the number of times multiple contiguous superblocks are required, requesting large areas of virtual memory is not uncommon for database systems. While CAS operations can't provide atomic allocation of multiple blocks, a sequence of superblocks can be allocated with an optimistic approach. Once a contiguous number of free superblocks is identified, the algorithm shown in Figure 5.4 tries allocating the blocks individually. For optimistic allocation, the `free_idx` is first swapped with 254 (`reserved_idx`) to signal that the superblock's state is temporary, as another core may allocate a free superblock required to fulfill the request during this phase. In this case, already reserved blocks are dropped before the core tries to find a new section of contiguously free superblocks. After successful allocation of the required number of blocks, the `reserved_idx` is overwritten with the calling cores ID. Depending on the superblock size, allocation of new superblocks is rare, reducing the occurrences of such races.

Deallocation. While identifying free superblocks during the deallocation of memory regions does not pose a performance overhead, returning those blocks causes uncertainty and incurs code complexity. Therefore, the current implementation avoids freeing superblocks once their state is set to a core's ID.

5.2.2 Superblock Manager

The superblock manager provides an interface to interact with the decentralized *VM buckets*. Each core accesses one bucket for all its virtual memory allocations. Lookup and deallocation of virtual memory follow the bucket containing the corresponding superblock.

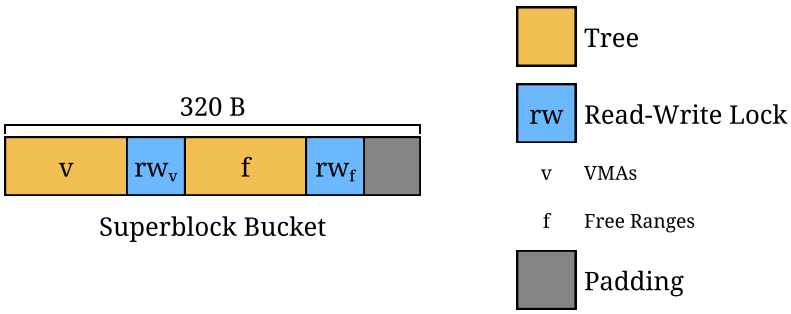


Figure 5.5: Data layout of one superblock bucket.

Data Structure. The superblock manager maintains an array of VM buckets. Each bucket contains, as depicted in Figure 5.5, two trees and a read-write lock for each to synchronize operations accordingly. Both trees are implemented as `boost::intrusive` sets, leaving memory management to the caller. The boost library bases its set on red-black trees, allowing logarithmic lookup time and fast rebalancing. While the

```
struct vm_range : public bi::set_base_hook<> {
    uintptr_t addr;
    u64 size;

    bool operator<(const vm_range& other) const {
        return addr < other.addr;
    }
};
```

Figure 5.6: Node of a free range set. The constructor has been omitted.

implementation and storage of VMAs within the decentralized design remains unmodified, major changes are made to storing free ranges. As seen in Figure 5.6 a node of a free range set stores the starting address of the range as well as its size. The free ranges are ordered by address rather than size. While this increases the complexity of finding a fitting range, merging on deallocation can be done with logarithmic complexity.

Initialization. To comply with OSv's original virtual memory implementation, an array of 65 superblock buckets is initialized. The first 64 entries correspond to the core ID and hold the state for superblock allocation. The remaining bucket is used for memory mappings outside the superblock area. We explicitly separate these functionalities as virtual memory outside the superblock region does not require superblock allocation. Instead, the bucket's free memory ranges are initialized with the entire virtual memory, except the superblock area. Therefore, the bucket can handle all requests independently of the superblock implementation.

Variable Range Allocation. Virtual memory can be mapped either at an arbitrary address chosen by the kernel or at a specific address requested by the caller. The first case is trivial, as the calling core will search the free ranges set of its VM bucket to find a suitable memory region. If the first region large enough to fulfill the request exceeds the required memory, the range is split, and the second half is used for allocation. As VM ranges are sorted by address, this allows editing the size of the node in place, thereby avoiding deletion and re-insertion, which possibly triggers rebalancing operations. Subsequently, a VMA is created for the allocated VM region and inserted into the bucket's VMA set. As this approach only touches the VM bucket designated to the calling core, contention on allocation is minimized.

Fixed Range Allocation. The decentralized design incurs complexity on allocation of specified memory regions. The validity of the requested range has to be verified, by iterating over intersecting superblocks. If the request can proceed, according to the rules detailed in Section 4.4, the virtual memory is allocated, and the corresponding VMA created and inserted in the VM bucket. Otherwise, the mapping request returns with an error.

Deallocation. On free, the VM range must be re-inserted into the corresponding free ranges set. Since the data structure is ordered by address, the previous entry can be found with logarithmic complexity, potentially allowing the freed VM range to merge into the previous one, thereby avoiding allocating and inserting a new node. Lastly, the adjacent memory region is examined to determine whether it can be merged.

6 Evaluation

In Chapter 3 we detail our design goals. We aim to provide an (1) low latency, highly scalable unikernel design, (2) with a low memory footprint, (3) tailored to database workloads (4) and capable of DBMS/VMS integration. We propose our design in Chapter 4 and provide implementation-level details for applying this design to the unikernel OSv in Chapter 5. This chapter evaluates whether the optimized OSv version meets the specified goals. Therefore we provide performance measurements for different memory utilization patterns and investigate boot time and memory consumption. Subsequently, we analyze performance for database workloads by comparing the efficiency of OLAP queries on different systems.

6.1 Testing Methodology

Competitors. OSv's original implementation provides the baseline performance for all measurements. We compare the baseline with our optimized OSv version and include the performance of a traditional Linux virtual machine when applicable.

Limiting Measurement Error. As single memory operations execute within a few CPU cycles, our benchmarks limit the measurement error by grouping multiple operations and calculating the average. This avoids distorting the results with noise caused by measurement operations. If non-deterministic results are observed, the experiment is repeated 10 times, and the mean and standard derivation are calculated.

Heap Allocator. We utilize jemalloc as memory allocator, as it shows exceptional performance for database workloads [6]. Applications running on Linux and the optimized OSv version are statically linked to jemalloc and utilize it as heap allocator. For the original OSv implementation, however, jemalloc is bound by the limited scalability of the virtual memory subsystem. Subsequently, the implemented allocation mechanisms, which avoid virtual memory operations, provide better performance for all synthetic workloads. Therefore, we avoid using a custom allocator for synthetic performance measurements on OSv's original implementation and rely on the unikernels native allocator.

6.2 Experimental Setup

6.2.1 Testbed

Machine. All experiments are performed on a single socket host system with an AMD EPYC 9654P processor (96 cores @ 2.4 GHz, 192 hardware threads) and 768 GiB of RAM. Each core has a 32KiB L1 instruction cache, a 32KiB L1 data cache, and a 1MiB L2 cache. The cores share 12 32MiB L3 caches. The host operates on Linux with kernel version 6.12.12. We employ the hypervisor qemu (version 9.2.0) for OSv and the Linux virtual machine. Linux performance is measured on a virtual machine with kernel version 6.1.96.

OSv Configuration. We compile OSv with `fs=ramfs` for all synthetic benchmarks. The optimized version of OSv utilizes the superblock allocator with a configured block size of 1 GiB. OSv starts with 60GB of RAM if not explicitly specified differently. The number of virtualized CPU cores OSv starts with varies for each boot, as depicted in the corresponding Figures.

6.2.2 Boot Time Measurement

While boot time was not a major goal of our design, significantly increasing startup times can impact feasibility for cloud workloads, where single instances are often spun up and down. Therefore, we include boot time analysis in our evaluation. To measure boot time, we compile OSv with a minimal C++ image, minimizing the initialization overhead caused by the application.

6.2.3 Memory Footprint Measurements

In Chapter 3, we detail how a low memory footprint is vital to address increasing database sizes. To compare the memory utilization of the original OSv version and our optimized design, we employ the kernel API to access the state of the physical memory allocator. Our benchmark measures the free memory after boot. Subsequently, it requests a number of memory regions and measures the free memory again. From these measurements, we calculate the memory overhead in relation to the number of requested bytes and present our findings in percent. We measure memory overhead for different allocation sizes by allocating the following sizes in bulk before taking the second measurement:

Small Allocations We request multiples of 256 B during one run and measure the memory footprint. The benchmark iterates over all multiples between 256 B and 4 KiB. This evaluates the memory overhead of small memory allocations.

Page Size Aligned Allocations Databases often utilize arena allocators [34, 12], therefore requesting multiple pages from the operating system. To measure the memory overhead of this allocation pattern, we request memory aligned to full pages.

To show the origin of memory overhead, we compare the memory efficiency of the original version of OSv and the optimized version with and without jemalloc as allocator.

6.2.4 Synthetic Performance and Scalability Benchmarks

To show the general performance of the implemented VMS for general workloads, we design and employ a custom benchmark suite. In this subsection, we first define abstract access patterns simulated by our synthetic benchmarks before enumerating how the access patterns are parameterized and what workload this simulates.

Access Patterns. The benchmark suite implements the following high-level access patterns that take a set of lambda functions to execute. We provide a small example for each pattern.

Bulk: All cores execute a specific function n times, storing the result in a local buffer. Once a core finishes, it runs a second function n times, each time with the corresponding parameter from the previously filled buffer. For bulk `allocate(4096)+free`, each core first allocates n pages and then frees each page again.

Pipeline: In the first stage of the pipeline pattern, each core iteratively fills a local buffer by executing a specific function n times before passing the buffer to the next thread. In the second stage, each core executes n functions with the corresponding value from the buffer as the parameter. For pipeline `allocate(4096)+free`, each core first allocates n pages and then frees all pages allocated by the preceding core.

Random: In each iteration, all cores execute a function for each parameter stored in a randomized subset of values from a global buffer and overwrite the values with new ones generated by a second function. The subsets change between every iteration. We independently measure the time for both functions and produce an average over all iterations and threads. For random `free+allocate(4096)`, each core frees n pages stored at random indices in the global buffer before storing the pointer to n newly allocated pages in the same place. In the next iteration, each core does the same for a different subset of pages from the global buffer.

Benchmarks. We run the following experiments on Linux, the original version of OSv, and the optimized version detailed in Chapter 4 and 5 and compare the results.

(1) Bulk: `mmap + munmap`
Many scalable heap allocators utilize core-local memory pools [16, 13]. These allocators exhibit a similar `mmap` pattern as simulated by this benchmark. It requests anonymous, 4 KiB mappings without requesting immediate allocation of physical memory. This experiment shows the performance and scalability of virtual memory allocation.

(2) Bulk: `allocate(4096);store + free`
To compare the maximum throughput of our design, we allocate a single page and write to it. This puts strain on the heap allocator and the system's virtual and physical memory allocator. Allocation policies differ between competing OSv versions. While the `malloc` implementation of the original OSv version is incapable of lazy allocation, jemalloc utilized the concept for newly allocated pages. Therefore, we write to (`store`) the page, potentially triggering a page fault and consequent physical allocation of each page. This benchmark additionally considers lookup time for Linux and the optimized OSv version due to these lazy allocations when utilizing jemalloc.

(3) Pipeline: `allocate(4096);store + free`
To simulate stream behavior, we measure pipeline performance. We provide ideal conditions for the original OSv implementation by choosing 4 KiB sizes due to its core local page pools. The argument for allocating and writing to the page remains identical to the previous benchmark.

(4) Random: `free + allocate(4096);store`
We provide a worst-case estimate for scalability by randomly allocating and deallocating memory across cores. This benchmark is designed to clash with the core-local virtual memory architecture of the optimized OSv version.

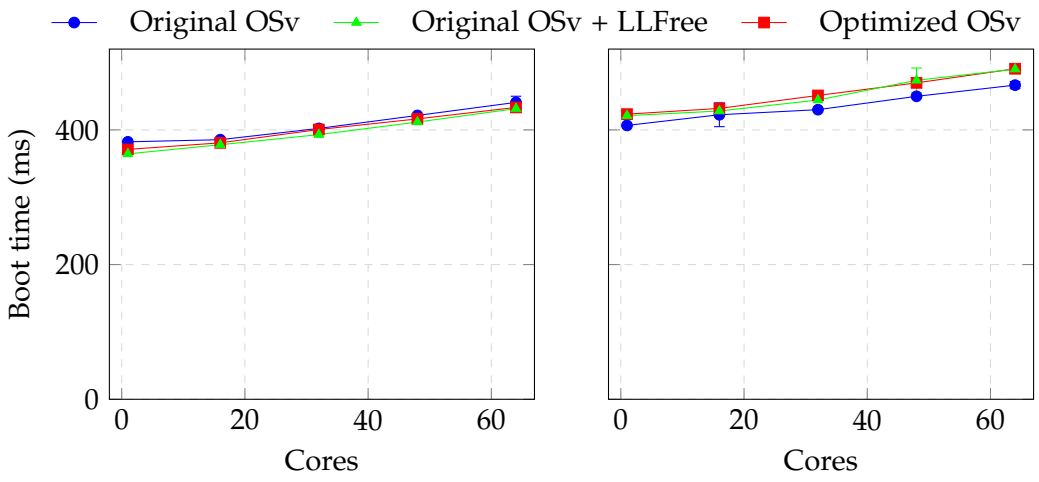
6.2.5 Database Workloads

We run the TPC-H benchmark on DuckDB [5] utilizing a database with a scale factor of 30 to measure how the unikernel design handles OLAP workloads. The corresponding data set is created prior to the benchmark and stored on a ZFS file system. Benchmarks are generally performed in memory on a warm database. The database is warmed up by running every query 11 times and discarding the first result. We thereby omit the noise caused by file I/O during the first query, using the remaining 10 query results to calculate the mean and standard derivation.

Due to scheduling issues in OSv, we utilize a modified DuckDB version, where worker threads are bound to a specific core. The modified DuckDB version is available in our benchmark suite on github[1]. While similar to prior synthetic benchmarks, the Linux and optimized OSv version harvest jemalloc performance, the original OSv version does not show better performance for all queries with its built-in allocator. Therefore, we present its performance with and without jemalloc.

6.3 Experimental Results and Analysis

6.3.1 Boot Time Measurement



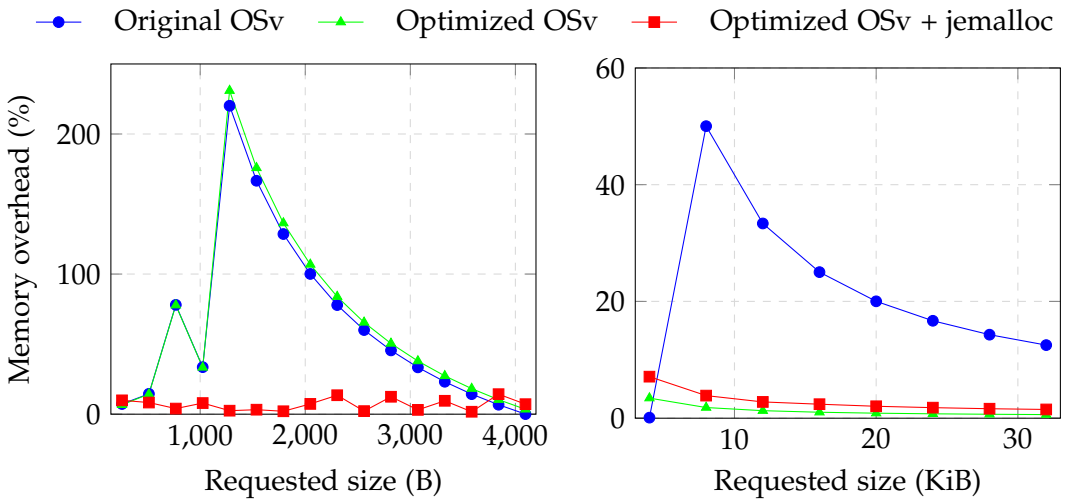
(a) 1GB RAM (b) 500GB RAM

Figure 6.1: Boot time comparison of different OSv versions per cores and main memory. The plot includes standard derivation if it exceeds 5 ms.

As shown in Figure 6.1, our design has no impact on boot time for small amounts of main memory passed to OSv while slightly increasing boot time for instances booted with large amounts of RAM. We include an OSv version with the original design but LLFree as physical memory allocator to show the origin of the minor performance loss during boot. Thereby, we can demonstrate that the minor boot time increase stems from the initialization of LLFree, as it has to iterate over the entire physical memory to mirror the system's allocated page frame state during initialization.

[1] `https://github.com/Martin-Lndbl/osv_benchmarks`

6.3.2 Memory Footprint Measurements



(a) Small allocations

(b) Page size aligned allocations

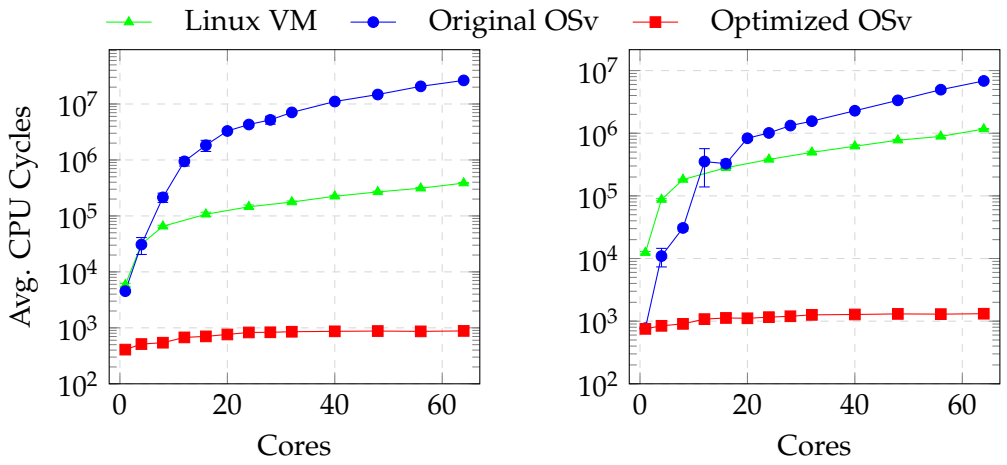
Figure 6.2: Memory overhead comparison of different OSv versions per requested memory size. The experiment produces deterministic results.

In Figure 6.2a, we see the memory efficiency of the small memory allocation mechanism in both the original and optimized versions of OSv. The comparison with the version utilizing jemalloc shows the significant memory efficiency boost possible by utilizing a suitable heap allocator.

Figure 6.2b shows the memory overhead of page size-aligned allocations for different OSv versions. The figure illustrates a clear improvement in the optimized OSv implementation. This is mostly due to the different approaches in tracking sizes of allocated memory regions. As the original version utilizes headers in front of the allocated memory, it may allocate an additional page frame. This occurs either because the requested alignment requires the returned memory region to start on page boundaries, or because the requested memory size is too close to the next page size to fit the additional bytes into the same page frames. Furthermore, we see that the native solution implemented in the optimized OSv version beats jemalloc regarding memory efficiency. This is unsurprising, as jemalloc utilizes the same endpoints to allocate its buffers that OSv's native allocator uses for each `malloc` request.

Note that the applied allocation of page-aligned sizes, which occurs often in database workloads, constitutes a worst-case scenario for the original OSv implementation, as every allocation requires an additional page for the header.

6.3.3 Synthetic Performance and Scalability Benchmarks



(a) Create virtual mapping

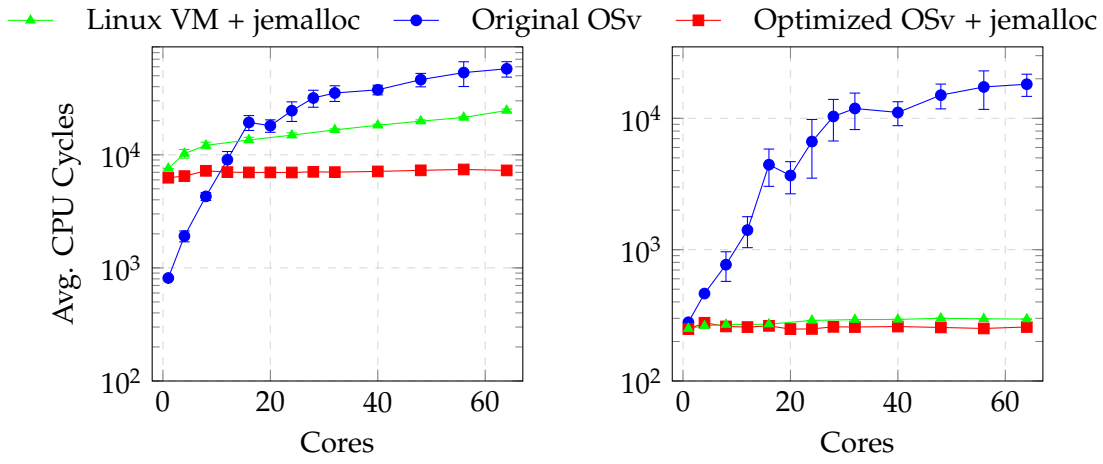
(b) Destroy virtual mapping

Figure 6.3: Bulk - `mmap + free`: The average speed of mapping and unmapping requests on different systems, including standard derivation.

(1) Bulk. We evaluate the scalability of the virtual memory subsystem by creating 20,000 VMAs in bulk, as specified in Subsection 6.2.4. Figure 6.3 compares the average performance during the creation and destruction of the virtual mappings.

Figure 6.3a and Figure 6.3b show our implementation's superior latency and scalability compared to the original OSv implementation and Linux. On creating the virtual mapping on a single core, we observe a ten-fold performance improvement compared to Linux. With higher concurrency, this gap increases due to the superior scalability of our design. The decentralization of state in the VMS allows all cores to allocate virtual memory and create VMAs in their own buckets, resulting in no contention for the simulated workload. Linux, on the other hand, does not scale well for memory mapping operations. The average number of CPU cycles required to create a mapping increases approximately by a factor of 1000 between 1 and 64 cores.

For the selected benchmark configuration, the original implementation of OSv requires around 900,000 CPU cycles for a single core and reaches over 30 million CPU cycles at 8 cores. The poor performance is due to architectural shortcomings of storing free VM ranges, leading to high latency to find a satisfactory VM region as well as global synchronization of the VMS. After reducing the number of created VMAs per core by 99%, the memory mapping operations still scale worse than its competitors, as shown in Figure 6.3.



(a) Allocation (b) Deallocation

Figure 6.4: Bulk - `allocate(4096);store + free`: The average allocation and deallocation performance on different systems, including standard derivation.

(2) Bulk. To evaluate the throughput of the overall system, we run the (2) bulk benchmark detailed in Subsection 6.2.4. Each core allocates and frees 200,000 pages.

Figure 6.4 compares the throughput of the competitors on a logarithmic scale. OSv's original design utilizes a physical memory allocator that relies on global synchronization. As expected, this locked approach results in the worst performance in this comparison. Although memory allocation of the original version does not consider VM allocation, the physical memory allocator exhibits a high degree of lock contention as the page pool capacity is insufficient to fulfill the entirety of allocation requests.

Like the original OSv implementation, jemalloc depends on core-local arenas for concurrent memory operations. This works well with the superblock design of the optimized OSv version. As jemalloc's core-local arenas ensure VM regions are mapped and unmapped by the same core, all operations on virtual memory can execute concurrently. Therefore, the optimized design of OSv scales very well for allocations (Figure 6.4a). Although jemalloc increases allocation throughput on Linux, the experiment shows that the underlying VMS does not scale well enough to provide the required VM mappings. This results in increased usage of CPU cycles for highly concurrent allocation patterns, as seen in Figure 6.4a.

The identical performance of our OSv version and Linux for deallocations, as depicted in Figure 6.4b, illustrates how jemalloc mitigates performance overhead by buffering freed memory. Therefore, no conclusions regarding scalability differences at deallocation between Linux and our unikernel version can be made.

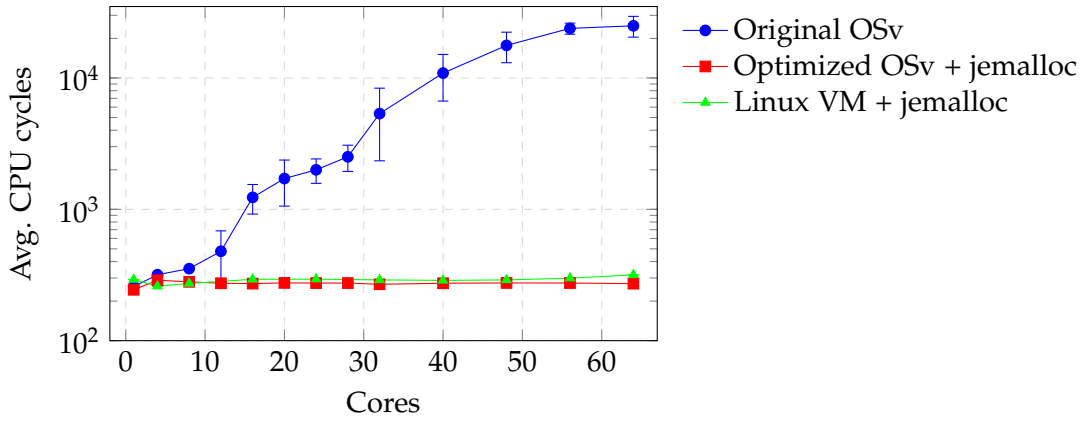


Figure 6.5: Pipeline - `allocate(4096);store + free`: Average free performance for pipeline workloads on different systems, including standard derivation.

(3) Pipeline. Moving towards more realistic workloads, we evaluate the (3) pipeline benchmark with 200,000 operations per stage. As the allocation phase of the benchmark is identical to the previously analyzed (2) bulk benchmark, we only evaluate deallocation performance.

Analyzing the results shown in Figure 6.5, we see better scalability for the original version of OSv than in the bulk benchmark (Figure 6.4b), even though the workload for each page batch is distributed between two cores. This is due to the fact that OSv utilizes garbage sinks to free memory allocated on another core. The memory is registered in the corresponding sink and the calling core returns without freeing any memory. A helper thread residing on the correct core will later pick up the memory from the garbage sink and free it properly. Therefore, the performance we see in Figure 6.5 is only a lower bound of the actual cycles required to free the memory in the original OSv version.

Unexpectedly, the performance at free remains unmodified for Linux and the optimized version of OSv. This is due to the internals of the utilized jemalloc allocator. When the workload changes, the heap allocator migrates arenas between cores to maximize memory locality. For the pipeline workload, this results in only core-local deallocations for both jemalloc and the underlying operating system. This property underlines the good fit of jemalloc for our design, as it effectively reduces contention caused by memory locality.

The pipeline benchmark was, however, insufficient to challenge the scalability design of our OSv version. Nevertheless, note that only the improvements made to the unikernel allow efficient utilization of mmap-based custom memory allocators, like jemalloc.

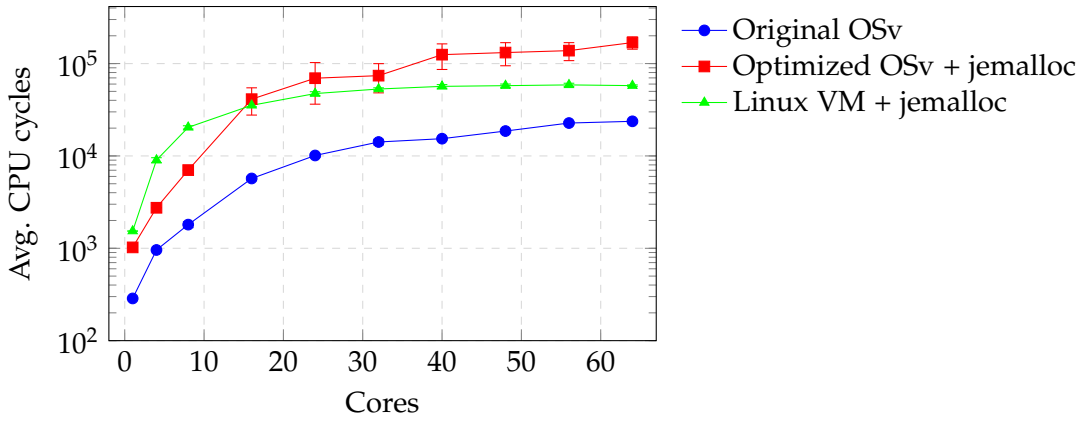


Figure 6.6: Random - `free + allocate(4096);store`: The average deallocation performance for randomized memory/core distribution on different systems, including standard derivation.

(4) Random. We run the random `free + allocate(4096);store` benchmark of our custom benchmark suite to test our design in highly shared workloads. In 10 iterations, each core frees 20,0000 random pages from a 2,000,0000-page buffer and refills the buffer by allocating new pages. As the allocation still happens core-local to each thread, we omit the allocation graph, as it does not contain new information. We show deallocation performance in Figure 6.6.

The figure shows the original OSv version with the best performance on deallocation. Performance on free is, in fact, not impacted by the core distribution for the large number of pages, since all free operations exceeding the page pools need to be synchronized on a global lock. However, this measurement is impaired by lazy deallocation, as the original OSv version utilizes garbage sinks.

Linux and the optimized OSv version share similar performance for deallocation (Figure 6.6). However, scalability diverges after reaching 32 cores, as OSv's VMS design focuses heavily on core locality. Unlike the (3) pipeline benchmark, the randomized memory pattern prevents jemalloc from migrating arenas to different cores, resulting in poor memory locality in jemalloc and the underlying VMS. This causes heavy contention of VM buckets in the optimized OSv version.

Linux, on the other hand, employs a maple tree for virtual memory areas that allows locking individual nodes instead of entire VMA buckets. Therefore, the design of the Linux kernel handles the highly shared workload simulated by the random benchmark better than our VMS design.

6.3.4 Database Workloads

We evaluate performance for OLAP workloads by running the TPC-H queries in DuckDB. Due to immense performance differences between the Linux virtual machine and the original OSv implementation, comparing each with the optimized OSv version is done separately.

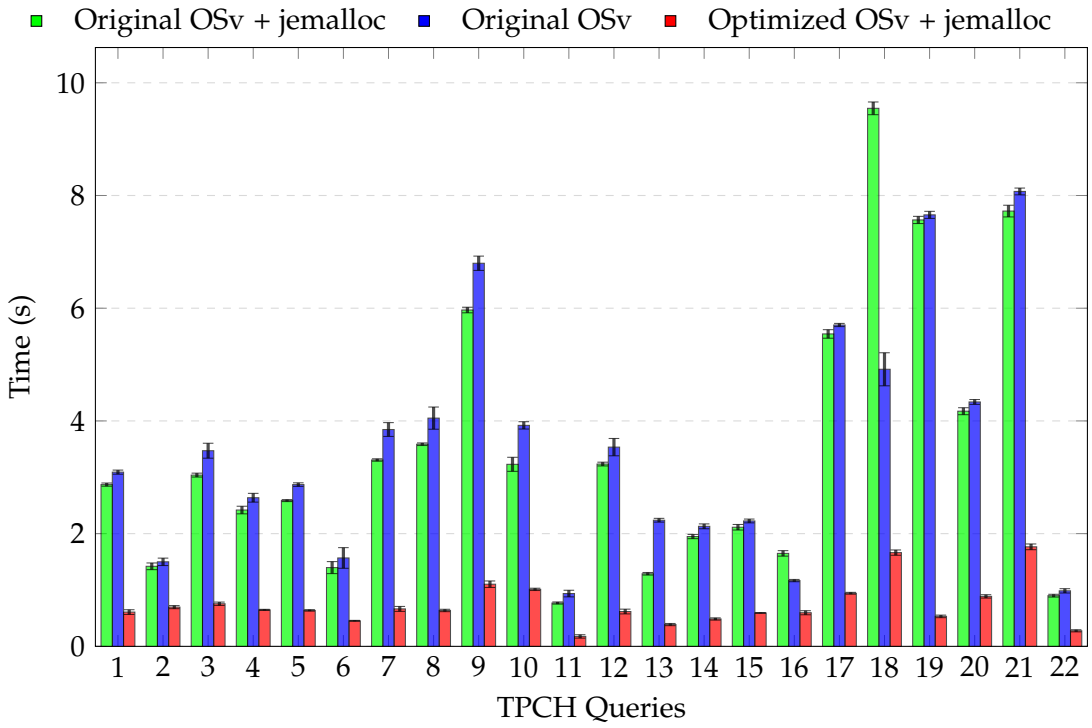


Figure 6.7: Mean performance of TPC-H queries with scale factor 30 on different OSv versions, including standard derivation.

Original OSv Comparison. The TPC-H performance comparison between the original OSv version and our design implementation is shown in Figure 6.7. We include the performance of the original version with and without jemalloc, as performance varies depending on the workload. The plot shows that the original version of OSv, regardless of the allocation mechanism, performs worse for database workloads compared to our optimized implementation. This is mostly due to the poor scalability of memory operations, especially memory mappings. With a quadratic mean of 4.2 seconds with and 4 seconds without jemalloc, it takes approximately 5 times as long as the optimized version with a quadratic mean of 0.8 seconds.

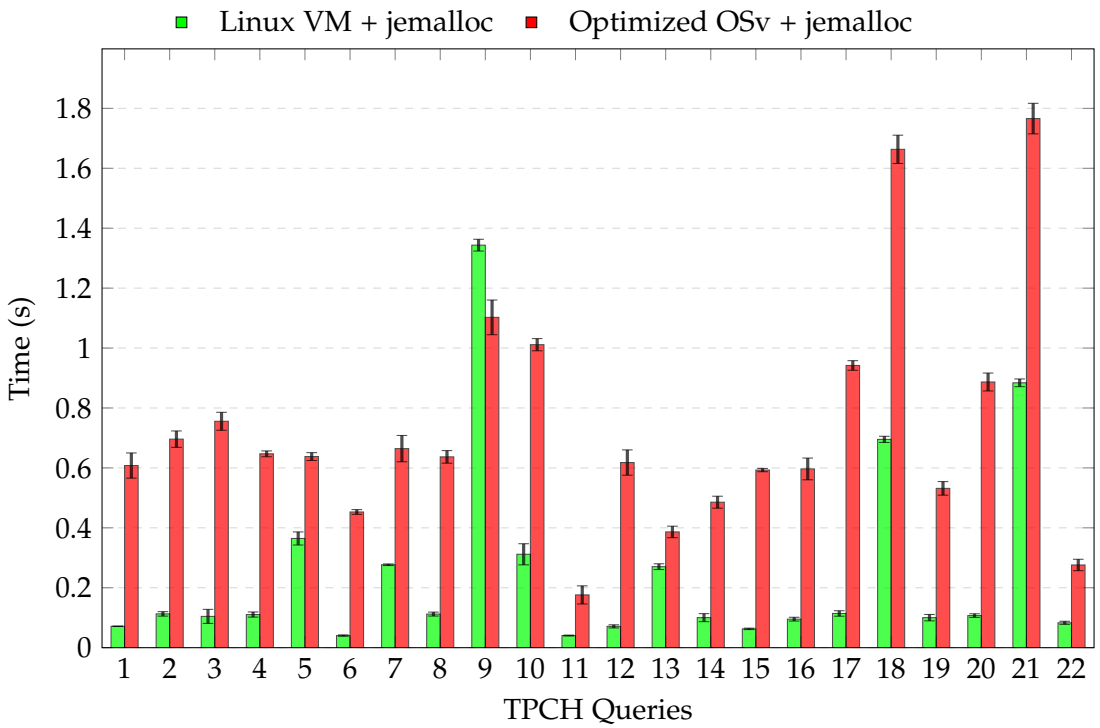


Figure 6.8: Mean performance of TPC-H queries with scale factor 30 on Linux and the optimized OSv version, including standard derivation.

Linux Virtual Machine Comparison. Our design achieves a quadratic mean runtime of 0.8 seconds, thereby exhibiting a performance improvement of 500% compared to the original OSv version. Figure 6.8 shows, however, that our design does not match or surpass Linux performance for most TPC-H queries. With Linux reaching a quadratic mean runtime of 0.4 seconds, our implementation requires on average double the time per query. We make the hypothesis that this performance loss is caused by OSv scheduling issues, as subsequent investigation shows low core utilization during query execution in OSv. This behavior is slightly improved by binding worker threads to a respective core, but not entirely mitigated.

7 Discussion

Static Memory Consumption. The superblock array detailed in Section 4.3 provides lock-free allocation of superblocks and functions as a hash table to look up VM buckets corresponding to virtual memory addresses. The array does, however, introduce considerable static memory overhead depending on the number of superblocks in the overall system and size of the atomic state value. Our optimized OSv version utilizes 1-byte state values and represents 32 TiB of virtual memory with 1 GiB large superblocks. The superblock array in this implementation causes a static memory overhead of 32 KiB. Assuming an implementation utilizes all 57 bits available on new CPUs [36] for superblock allocation, the superblock array grows to a size of 128 MiB, significantly increasing the minimal amount of RAM required to boot the unikernel. Further, allowing more than 253 cores doubles this overhead, as 2-byte state values are required to store the core ID.

Decentralized VMA Storage. Our design proposes storing VMAs alongside free memory ranges in decentralized VM buckets. While this does not incur large synchronization overhead during allocation and free, as synchronization on bucket boundaries is also necessary for VM range operations, it introduces contention for lookups. It further disallows specific memory allocation across distinct buckets. The design thereby exhibits limited scalability for page faults and constrains applications that rely heavily on memory mappings with explicit virtual memory ranges.

Underdevelopment. We believe that OSv displays promising potential for cloud-native database systems. However, compared to Linux, which has thousands of contributors, the unikernel lacks maturity. Within the last year, OSv had less than 5 contributors[1]. This underdevelopment can lead to correctness issues. While generating databases for TPC-H queries on DuckDB, we encountered runtime errors caused by data races in both the original and optimized versions of OSv. For mainstream adoption, unikernels require further reliability improvements.

Single Process Limitation. By design, unikernels pose certain limitations on the variety of applications they can host. Most prominently, no address space separation induces a single-process constraint. This excludes large databases like PostgreSQL [31] that rely on spawning multiple processes from being supported on the unikernel architecture.

[1] `https://github.com/cloudius-systems/osv/graphs/contributors?from=3%2F2%2F2024`

8 Related Work

We categorize related work in two areas: (a) altering operating systems to allow DBMS integration (b) general improvements made to the VMS.

8.1 DBMS/OS co-design

VmCache [18] implements the ExMap kernel module to allow WAL for memory-mapped databases by controlling page eviction. It further implements measures increasing the performance of the VMS, including batched TLB shootdowns. Kreon [28] goes a step further by implementing a `mmap` alternative (`kmmap`) tailored to the needs of the key-value store. The presented kernel module improves traditional `mmap` scalability by splitting memory into independent banks, similar to the superblock design. It furthermore limits contention within a bank by fine-grained locking. Both designs rely on kernel modules, making them difficult to maintain for progressing kernel versions. Kernel modification furthermore disallows customization for single applications.

Skiadopoulos et al. [32] present DBOS to manage database clusters in a cloud environment. The authors propose the utilization of a microkernel without sophisticated memory management as an underlying operating system and move memory management to the DBMS level. The single instance philosophy of unikernels and the general idea of a cluster manager for multiple database applications make DBOS fundamentally orthogonal to the concepts of our work.

Libdbos [40] and Aquila [27] employ dune [1] to utilize a modified Linux kernel and expose functionality to the application layer. While Aquila focuses on fast mmio with custom kernel paths in general, the authors of Libdbos propose using a privileged kernel-bypass explicitly for DBMS co-design. Libdbos provides similar flexibility in adjusting kernel functionality as presented in this work, however, the general approach of applications bypassing the kernel poses lower security guarantees. Furthermore, direct modification of the VMS in the Linux kernel is complex due to its size and entanglement with other kernel subsystems. Similar to kernel modules, the approach requires dune and kernel modifications to be maintained over time. We argue that unikernels provide a more suitable base for kernel modifications due to their simplicity and safety provided by the hypervisor as well as narrowed attach surface [35].

COD [10] shares the general goal of integrating the OS and DBMS. It provides a proof of concept for information exchange by designing and implementing OS-level interfaces that can be used by the DBMS to place scheduling hints. The approach incurs Linux kernel modification, thereby suffering from the complex code base and limited flexibility of a general purpose OS.

Analogous to our approach, Giceva et al. [11] present a design for customized OS support for data processing, that exposes control over customized kernel functionality in different abstraction levels to the DBMS. The authors envision a layered approach, consisting of: (1) A customized lightweight kernel that exposes desired interfaces on a system level. (2) A library OS running in user space to expose kernel functionality to the application and abstract complex protocols. (3) A runtime system providing high-level interfaces to the DBMS. Although the proposed design promises a significant performance boost combined with enabling the DBMS to influence VM and scheduling decisions, the privilege isolation in traditional operating systems still incurs context switches, limiting the potential performance gain. Address space isolation further limits the flexibility the kernel interfaces can provide, potentially requiring changes on the microkernel level if the DBMS requires functionality not exposed by current implementation.

8.2 VM Subsystem improvements

Clements et al. [3] propose utilizing a tree structure for VMAs that allows lock-free lookups. The Bonsai tree reduces contention in the VMS as page faults can execute concurrently. Building on this earlier work, the authors present RadixVM [2]. The novel VMS improves scalability for non-overlapping address ranges by implementing a radix tree structure to store metadata. This allows fine-grained locking for VMA creation and deletion as well as lock-free lookups. The approach is orthogonal to our design as the superblock allocator design focuses on the scalable identification of free VM ranges, with decentralized metadata storage as a byproduct. By implementing a radix tree in OSv, the scalability of our design may be improved further.

Several solutions to improve the VMS based on kernel modules have been proposed [8, 33]. The optimization techniques include batching TLB shootdowns, improving page eviction by bypassing the page cache as well as buffering recently freed pages on the application level. Apart from batched TLB shootdowns, the measurements taken mostly react to insufficient scalability of the Linux kernel implementation and are, therefore, analogous to our design. As the modules do not enable page eviction control, let alone control over kernel data structures, they are unsuitable for memory-mapped databases.

FastMap [29] proposes an optimized design for the mmio path. The custom VMS implements measures to increase efficiency and scalability compared to Linux. The main approach separates clean and dirty pages, but other optimizations, such as batched TLB shootdowns, are included in the design. FastMap does not provide extensive control over the VMS to the application, making our work orthogonal.

The user-space framework UMap [30] bypasses the kernel to generate VM mappings in user space. As page faults are handled utilizing `userfaultfd`, the approach suffers from context-switching overhead. Like most work in this area, UMap does not fix fundamental issues of the Linux VMS regarding database systems.

9 Conclusion

The mismatch between operating systems and database engines motivates architectural changes on both sides. Traditionally, the operating system multiplexes applications for fair and efficient resource distribution. Thereby, intricate knowledge of the DBMS for scheduling and memory management is ignored. Database systems often implement their own abstractions to bypass this issue and avoid connected performance penalties.

To solve this mismatch, unikernel co-design for cloud databases has been proposed. This work contributes to this goal by presenting a general kernel API specification for VMS/DBMS integration in unikernels, combined with a scalable virtual memory subsystem design with a low memory footprint tailored to database workloads.

We utilize LLFree as page frame allocator, providing optimal performance, scalability, and anti-fragmenting behavior, while requiring higher abstraction layers to accommodate the limited width of allocation requests. We examine and optimize the virtual memory subsystem's scalability by introducing a decentralized design for VM allocation and bookkeeping. We untangle the VMS logic and components for better modularity, facilitating easy modification of individual data structures and procedures on an application level.

By exposing VMS logic to the DBMS, memory management is treated as a library feature, maintaining default functionality while providing the freedom to change and intercept logic both on a coarse and fine-grained scale.

For our implementation in OSv, the experiments detailed in Chapter 6 show significant scalability improvements compared to the original unikernel version while simultaneously reducing the memory footprint. Our design also performs better than Linux in our synthetic benchmark suite. Even without utilizing the kernel API, our design significantly narrows the performance gap between Linux and the unikernel for database workloads.

The source code of the modified OSv version, as well as the benchmark suite utilized in Chapter 6 are available at: `https://github.com/Martin-Lndbl/osv_controller`.

10 Future Work

Although our design significantly improves the scalability of the VMS, several approaches remain unexplored. We outline further efforts in two categories: (a) further improving the VMS, including the implementation of new features (b) necessity of kernel integration in other subsystems

10.1 VM Optimizations

TLB Shootdowns. Related work proposes batched [18, 8, 29] or targeted [2] TLB shootdowns. As global TLB shootdowns remain a bottleneck in unoptimized unikernel designs, tackling this issue may result in better performance on unmapping virtual memory. Our design even increased the severity of this issue by increasing the frequency of memory mappings.

Data Structure Considerations. The currently used red-black trees for storage of VMAs and free memory ranges do not allow lock-free page faults, as modifying the data structure while traversing it can result in undefined behavior. The scalability of our design could be improved by utilizing bonsai [3], radix [2], or maple [22] trees instead.

Page Replacement. Page replacement is usually not implemented in unikernels [26, 37]. In the context of database systems, however, enabling hardware-assisted buffer management comes with several performance benefits over the traditional system call-driven buffer manager [4, 18, 14]. The current kernel API allows DBMS specific implementations of this concept. However, providing an optimized general interface for this functionality could decrease complexity on the database level without losing flexibility or functionality.

10.2 DBMS/OS Co-design

Leis and Dietrich [19] identify 4 hardware primitives that can be co-designed with the DBMS: (1) DBMS-Aware scheduling could make user-level scheduling obsolete. As kernel-level threads in unikernels inherit the performance benefits of the single address space, having all threads in the same place takes advantage of kernel features like preemption without suffering the overhead of context switches. Chapter 6 facilitates the

necessity for scheduling optimizations in OSv. (2) This thesis outlines the benefits of VMS/DBMS integration and presents a solution leveraging a kernel API. (3) Exposing state information about the system's memory to the hypervisor enables further performance benefits, like memory ballooning improvements. (4) By giving direct access to networking or NVMe queues to the database layer promises superior performance due to avoided indirection. Exposing the queues to the DBMS further allows dynamically choosing between low-latency polling and low-power-consumption interrupt policies depending on the workload.

Abbreviations

DBMS database management system

OS operating system

VMA virtual memory area

OOM out-of-memory

CPU central processing unit

MMU memory management unit

SMP symmetric multi-processing

TLB translation lookaside buffer

CAS compare and swap

API application programming interface

RAM random access memory

WAL write-aheah logging

mmio memory-mapped I/O

VM virtual memory

VMS virtual memory subsystem

OLAP online analytical processing

List of Figures

List of Tables

Bibliography

[1] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. "Dune: Safe User-level Access to Privileged CPU Features." In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 335–348. ISBN: 978-1-931971-96-6.

[2] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. "RadixVM: scalable address spaces for multithreaded applications." In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 211–224. ISBN: 9781450319942. DOI: 10.1145/2465351.2465373.

[3] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. "Scalable address spaces using RCU balanced trees." In: *SIGARCH Comput. Archit. News* 40.1 (Mar. 2012), pp. 199–210. ISSN: 0163-5964. DOI: 10.1145/2189750.2150998.

[4] A. Crotty, V. Leis, and A. Pavlo. "Are You Sure You Want to Use MMAP in Your Database Management System?" In: *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022.

[5] DuckDB. *DuckDB*. URL: https://github.com/duckdb/duckdb (visited on 02/10/2025).

[6] D. Durner, V. Leis, and T. Neumann. "On the Impact of Memory Allocation on High-Performance Query Processing." In: *Proceedings of the 15th International Workshop on Data Management on New Hardware*. SIGMOD/PODS '19. ACM, July 2019, pp. 1–3. DOI: 10.1145/3329785.3329918.

[7] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. "Dark silicon and the end of multicore scaling." In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 365–376. ISBN: 9781450304726. DOI: 10.1145/2000064.2000108.

[8] B. Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale. "DI-MMAP–a scalable memory-map runtime for out-of-core data-intensive applications." In: *Cluster Computing* 18.1 (Mar. 2015), pp. 15–28. ISSN: 1386-7857. DOI: 10.1007/s10586-013-0309-0.

[9] F. Funke, A. Kemper, and T. Neumann. "Compacting transactional data in hybrid OLTP&OLAP databases." In: *Proc. VLDB Endow.* 5.11 (July 2012), pp. 1424–1435. ISSN: 2150-8097. DOI: 10.14778/2350229.2350258.

[10] J. Giceva, T. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. "COD: Database / operating system co-design." English. In: Publisher Copyright: © 2013 Conference on Innovative Data Systems Research (CIDR). All rights reserved.; 6th Biennial Conference on Innovative Data Systems Research, CIDR 2013 ; Conference date: 06-01-2013 Through 09-01-2013. 2013.

[11] J. Giceva, G. Zellweger, G. Alonso, and T. Rosco. "Customized OS support for data-processing." In: *Proceedings of the 12th International Workshop on Data Management on New Hardware*. DaMoN '16. San Francisco, California: Association for Computing Machinery, 2016. ISBN: 9781450343190. DOI: 10.1145/2933349.2933351.

[12] Google. *LevelDB*. URL: https://github.com/google/leveldb (visited on 02/10/2025).

[13] Google. *tcmalloc*. URL: https://github.com/google/tcmalloc (visited on 02/26/2025).

[14] G. Graefe, H. Volos, H. Kimura, H. Kuno, J. Tucek, M. Lillibridge, and A. Veitch. "In-memory performance for big data." In: *Proc. VLDB Endow.* 8.1 (Sept. 2014), pp. 37–48. ISSN: 2150-8097. DOI: 10.14778/2735461.2735465.

[15] J. Gray. "Notes on Data Base Operating Systems." In: *Operating Systems, An Advanced Course*. Berlin, Heidelberg: Springer-Verlag, 1978, pp. 393–481. ISBN: 3540087559.

[16] jemalloc. *jemalloc*. URL: https://github.com/jemalloc/jemalloc (visited on 02/26/2025).

[17] KeyDB. *KeyDB*. URL: https://github.com/Snapchat/KeyDB (visited on 02/10/2025).

[18] V. Leis, A. Alhomssi, T. Ziegler, Y. Loeck, and C. Dietrich. "Virtual-Memory Assisted Buffer Management." In: *Proc. ACM Manag. Data* 1.1 (May 2023). DOI: 10.1145/3588687.

[19] V. Leis and C. Dietrich. "Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware." In: *Proc. VLDB Endow.* 17.8 (Apr. 2024), pp. 2115–2122. ISSN: 2150-8097. DOI: 10.14778/3659437.3659462.

[20] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. "LeanStore: In-Memory Data Management beyond Main Memory." In: Apr. 2018, pp. 185–196. DOI: 10.1109/ICDE.2018.00026.

[21] J. J. Levandoski, P.-Å. Larson, and R. Stoica. "Identifying hot and cold data in main-memory databases." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, pp. 26–37. DOI: 10.1109/ICDE.2013.6544811.

[22] Linux. *Linux*. URL: https://github.com/torvalds/linux (visited on 12/23/2024).

[23] LLFree. *LLFree-c*. URL: https://github.com/luhsra/llfree-c (visited on 11/18/2024).

[24] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. "Unikernels: library operating systems for the cloud." In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167.

[25] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. "Unikernels: library operating systems for the cloud." In: *SIGARCH Comput. Archit. News* 41.1 (Mar. 2013), pp. 461–472. ISSN: 0163-5964. DOI: 10.1145/2490301.2451167.

[26] OSv. *OSv*. URL: https://github.com/cloudius-systems/osv (visited on 11/18/2024).

[27] A. Papagiannis, M. Marazakis, and A. Bilas. "Memory-mapped I/O on steroids." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 277–293. ISBN: 9781450383349. DOI: 10.1145/3447786.3456242.

[28] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. "An Efficient Memory-Mapped Key-Value Store for Flash Storage." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: Association for Computing Machinery, 2018, pp. 490–502. ISBN: 9781450360111. DOI: 10.1145/3267809.3267824.

[29] A. Papagiannis, G. Xanthakis, G. Saloustros, M. Marazakis, and A. Bilas. "Optimizing Memory-mapped I/O for Fast Storage Devices." In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 813–827. ISBN: 978-1-939133-14-4.

[30] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale. "UMap: Enabling Application-driven Optimizations for Page Management." In: *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 2019, pp. 71–78. DOI: 10.1109/MCHPC49590.2019.00017.

[31] Postgres. *PostgreSQL 17.4 Documentation*. URL: https://www.postgresql.org/docs/17/index.html (visited on 03/01/2025).

[32] A. Skiadopoulos, Q. Li, P. Kraft, K. Kaffes, D. Hong, S. Mathew, D. Bestor, M. Cafarella, V. Gadepally, G. Graefe, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, L. Suresh, and M. Zaharia. "DBOS: a DBMS-oriented operating system." In: *Proc. VLDB Endow.* 15.1 (Sept. 2021), pp. 21–30. ISSN: 2150-8097. DOI: 10.14778/3485450.3485454.

[33] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom. "Efficient Memory-Mapped I/O on Fast Storage Device." In: *ACM Trans. Storage* 12.4 (May 2016). ISSN: 1553-3077. DOI: 10.1145/2846100.

[34] SQLite. *SQLite*. URL: https://www.sqlite.org/malloc.html (visited on 02/10/2025).

[35] J. Talbot, P. Pikula, C. Sweetmore, S. Rowe, H. Hindy, C. Tachtatzis, R. Atkinson, and X. Bellekens. "A Security Perspective on Unikernels." In: *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. 2020, pp. 1–7. DOI: 10.1109/CyberSecurity49315.2020.9138883.

[36] A. Tanenbaum and H. Bos. *Modern Operating Systems Global Edition*. Pearson Deutschland, 2014, pp. 179–199. ISBN: 9781292061429.

[37] Unikraft. *Unikraft*. URL: https://github.com/unikraft/unikraft/tree/staging (visited on 03/01/2025).

[38] L. Wrenger, F. Rommel, A. Halbuer, C. Dietrich, and D. Lohmann. "LLFree: Scalable and Optionally-Persistent Page-Frame Allocation." In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 897–914. ISBN: 978-1-939133-35-9.

[39] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan. "Performance analysis of NVMe SSDs and their implication on real world databases." In: *Proceedings of the 8th ACM International Systems and Storage Conference*. SYSTOR '15. Haifa, Israel: Association for Computing Machinery, 2015. ISBN: 9781450336079. DOI: 10.1145/2757667.2757684.

[40] X. Zhou, V. Leis, J. Hu, X. Yu, and M. Stonebraker. "Practical DB-OS Co-Design with Privileged Kernel Bypass." In: *Proceedings of the ACM on Management of Data* 3 (Feb. 2025), pp. 1–27. DOI: 10.1145/3709714.