Memory Matters: Load-Time Deduplication for Unikernels

Gaulthier Gain
University of Liège
Belgium

Benoît Knott
University of Liège
Belgium

Cyril Soldani
University of Liège
Belgium

Laurent Mathy
University of Liège
Belgium

Abstract

Unikernels offer strong isolation and performance benefits over traditional virtual machines, but their specialized, statically linked design complicates memory deduplication—particularly in multi-tenant environments. Traditional approaches like Kernel Samepage Merging (KSM) struggle with convergence delays, high CPU usage, and unpredictable memory savings. Dynamic linking improves memory reuse but compromises performance, simplicity, and security. We present Spacer-SLT, a novel load time deduplication mechanism that eliminates the need for memory scanners. Spacer-SLT extracts common libraries into a shared pool and uses a custom Firecracker-based loader to enable instant, deterministic memory deduplication at launch time. Unlike KSM, it introduces no runtime overhead and maintains the benefits of static linking, including performance and reduced attack surface.

Our experiments show that Spacer-SLT achieves superior memory reduction with lower overhead than traditional memory deduplication scanners and dynamically linked alternatives. Moreover, it ensures consistent performance under high concurrency—something runtime deduplication methods fail to guarantee. Spacer-SLT is fully compatible with ASLR and available as open source on GitHub [25].

CCS Concepts

• **Software and its engineering → Virtual machines**.

Keywords

Unikernels, Virtualization, Memory Deduplication, Cloud Computing, FaaS, Alignment

ACM Reference Format:
Gaulthier Gain, Benoît Knott, Cyril Soldani, and Laurent Mathy. 2025. Memory Matters: Load-Time Deduplication for Unikernels. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3772052.3772247

SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/2025/11
https://doi.org/10.1145/3772052.3772247

1 Introduction

Among the technological advances in cloud computing, unikernels [37–39, 46, 50, 58, 81, 83] are gaining traction for a variety of application domains. Unikernels follow a library operating system approach [4, 19, 42, 63, 69, 88], resulting in several advantages over traditional Virtual Machines (VMs) and containers [35, 36, 46, 48]. Unlike these general-purpose environments, unikernels are designed to be lightweight and specialized, with a single address space and only the necessary libraries and drivers. This streamlined architecture allows unikernels to boot rapidly, often in milliseconds, making them ideal for scenarios requiring rapid autoscaling and responsive scaling to handle increases in the workload [13, 43, 56].

Unikernel specialization offers many advantages, but it also introduces challenges—particularly in memory efficiency at scale [26, 27, 78]. Each unikernel is statically tailored for a specific purpose, with a compact layout and a tightly integrated set of libraries. As a result, even if multiple unikernels use the same libraries, they produce distinct binary images. This occurs because libraries are compacted within each instance. This becomes problematic in multi-tenant environments, where many unikernels run concurrently on the same server. Since most memory pages are instance-specific, only a small number of pages can be merged using memory deduplication—a technique that merges identical memory pages to reduce overall memory consumption. Consequently, while individual unikernels may have a small memory footprint, deploying hundreds or thousands of them can lead to unexpectedly high memory consumption—even with a background deduplication scanner like the Kernel Samepage Merging (KSM) daemon [6].

This limitation has already been addressed by Spacer [27] which leverages library alignment: placing libraries at fixed, absolute addresses to increase the likelihood of identical page layouts across unikernels. This significantly improves deduplication efficiency. Experiments with Unikraft [36] have shown up to a 3× reduction in memory usage with Spacer and KSM, compared with other configurations. However, Spacer still depends on runtime memory deduplication scanners, such as KSM, which present fundamental trade-offs: quiet modes are CPU-friendly but slow to converge, often missing sharing opportunities. Aggressive modes offer faster convergence but increase CPU load [27, 85] and may be overwhelmed by high-density and bursty serverless workloads. In such scenarios, KSM fails to provide uniform and timely memory reduction, leading to unfair memory sharing across instances. Additionally, deduplication mechanisms based on CoW (Copy-on-Write) semantics have been shown to expose timing side channels [3, 10, 30, 62, 76, 77, 86], undermining isolation guarantees in multi-tenant environments.

Another alternative is to adopt a dynamically linked model, where multiple unikernels share dynamic libraries at runtime. While this approach can significantly reduce memory usage, it compromises the immutability and simplicity that define unikernels. It also introduces runtime overhead and added complexity. This ultimately undermines key design goals such as performance and security [36, 41].

These limitations highlight the need for a better approach that preserves the core principles of unikernels while still enabling isolation and efficient memory reuse. To this end, we introduce Spacer-SLT (*Share at Load Time*): a novel mechanism (based on Spacer [27]) that enables deterministic memory sharing by mapping identical pages from different unikernel instances to the same memory frames at load time. Unlike prior solutions, Spacer-SLT ensures immediate memory sharing, eliminating the CPU cost and timing convergence trade-offs of deduplication scanners. It also maintains a fully static and immutable build model, avoiding the issues associated with dynamic libraries, such as symbol resolution [18], greater attack surface, and challenges with portability and deployment complexity [15, 17]. Spacer-SLT is implemented via a dedicated toolchain and a custom loader based on Firecracker [1]. Spacer-SLT also supports ASLR (Address Space Layout Randomization) for enhanced security, while still preserving deduplication efficiency. We evaluate Spacer-SLT across multiple configurations in both homogeneous and heterogeneous deployment scenarios. Results show that it delivers faster, fairer, and lower-overhead memory reduction, outperforming both runtime deduplication and dynamic linking approaches in serverless-like workloads.

Our main contributions are the following: (i) We extend Spacer with a toolchain and a custom loader based on Firecracker. This enables memory deduplication to occur at load time, rather than the slower, unpredictable, and more CPU-intensive runtime process. (ii) We present a comprehensive performance evaluation that highlights the limitations of the conventional method of loading unikernels and using KSM compared to our proposed approach.

2 Background

This section summarizes unikernels, detailing how they are built and loaded via a type-2 hypervisor. It also introduces Spacer, a tool leveraging library alignment to improve memory sharing in unikernels.

2.1 Unikernels

Unikernels represent a paradigm shift in operating system design, focusing on extreme specialization for running single applications or services. At their core, unikernels leverage a library operating system (libOS) concept, where an application is tightly integrated with the underlying kernel. In general, they are compiled to include only the necessary OS functionality, such as required system calls and drivers, forming a single and immutable executable image. The application code and libraries are compiled into a single statically linked image containing all the components necessary for seamless execution. When loaded, the hypervisor allocates virtual memory to create a dedicated space for the unikernel. It parses the ELF file, maps segments into memory with proper protections, and interacts with the unikernel using the Kernel-based Virtual Machine [61] (KVM) API to manage low-level hardware operations.

2.2 Memory Deduplication Scanner

The Linux Kernel Samepage Merging [6] (KSM) daemon identifies and deduplicates identical memory pages by mapping them to a single physical frame and marks them Copy-on-Write (CoW). KSM uses two red-black trees: the stable tree, for merged pages to optimize future lookups, and the unstable tree, for unchanged pages awaiting re-evaluation. The use of CoW semantics on deduplicated memory pages has been shown to introduce timing side-channel vulnerabilities, allowing malicious VMs to detect or infer the presence of shared pages with other VMs [3, 10, 30, 62, 76, 77, 86]. By measuring access latencies and triggering CoW faults (via writable pages), attackers can exploit these timing differences to extract sensitive information across isolation boundaries. KSM's behavior can be manually configured with several parameters [6, 20].

2.3 Aligning Libraries with Spacer

While unikernels deliver outstanding performance and have numerous benefits, their specialized nature introduces two main challenges when different unikernels include varying sets of libraries. In such cases, even common libraries may be placed at different memory locations across instances. These challenges are: (a) Page misalignment, making pages appear different (b) In addition, it makes cross-reference addresses different in instructions such as `CALL` or `LEA`. Both issues cause each unikernel to generate unique pages (even if they share the same libraries). These two limit the effectiveness of memory deduplication between unikernels.

The Spacer [27] page alignment tool was designed to mitigate those issues. The overall idea behind Spacer is quite simple: align libraries by placing them at absolute addresses (page aligned) that are common to all unikernel instances to increase the potential sharing between them. Spacer is a standalone tool that analyzes all unikernels in a workspace (a folder that contains several unikernels) and builds a global map that assigns a unique absolute address to each library. The tool generates a custom linker script for each unikernel. This script is then used to relink the unikernel image to use the assigned memory addresses. Each unikernel contains multiple sections, with some sections representing libraries that include their own code (`.text`) and read-only data (`.rodata`). When a library is not used in a particular unikernel, a gap (pages filled with zeros) is inserted in its place (through linker instructions). Spacer can perform link time ASLR to enhance unikernel security. This is achieved by employing a "trampoline table" for each library. These tables contain problematic instructions, such as `CALL` or `LEA`, that use addresses from other sections or libraries, allowing identical libraries to be used at different addresses across executions.

3 Problem Statement

Aligning libraries increases the number of identical pages, facilitating identification and merging by a runtime memory deduplication daemon like KSM. This results in notable memory savings when multiple instances sharing common libraries are active on the same machine. However, this method relies on a background memory scanner, which introduces inherent issues due to the scanning and

merging process that we aim to eliminate with our approach. The main limitations of KSM are: **§1 KSM convergence time**: KSM offers memory deduplication, but its effectiveness with ephemeral unikernels presents a double-edged sword. While a conservative strategy reduces CPU usage, it takes longer to identify duplicates. This significantly hinders KSM's ability to consolidate memory before short-lived unikernels terminate, leading to missed opportunities for memory savings. An aggressive KSM configuration is not a perfect solution either. While it speeds up deduplication, the rapid arrival of many unikernels can overwhelm KSM. This can still lead to missed sharing opportunities, limiting memory benefits. **§2 KSM performance (CPU)**: An aggressive KSM mode significantly reduces the time to consolidate memory but comes at the cost of higher CPU usage. KSM uses more CPU cycles to scan and merge identical pages, which negatively affects overall system performance. **§3 KSM variability and load sensitivity**: KSM's deduplication effectiveness is not consistent. Memory reduction depends heavily on the system load and timing of instance launches. Under low load, KSM can achieve substantial savings; however, as concurrency increases, its merging capability deteriorates, resulting in higher memory usage. This non-determinism can make resource provisioning unpredictable, particularly in multi-tenant environments. **§4 KSM performance (I/O and TLB[1])**: Before being merged by KSM into a single read-only (or copy-on-write) frame, identical pages are first allocated and then mapped to different physical frames. This requires unnecessary I/O and copies to allocate and fill frames. In addition, the merging process locks the page tables and issues TLB shootdowns (TLB flushes), which can increase memory access latency and degrade system performance. **§5 KSM writable pages**: Relying on memory deduplication for writable pages is vulnerable to memory disclosure attacks [3, 10, 30, 62, 76, 77, 86], which are based on a difference in write access time on the deduplicated memory pages that are recreated by CoW. Moreover, deduplicating writable pages can also degrade performance [27]. It is possible to modify this behavior, but it requires either modifying the underlying memory scanner or the hypervisor (i.e., the `madvise()` may be applied only to read-only pages for KSM). **§6 Spacer size (ELF inflation)**: Although alignment does not significantly increase image size, Spacer misses the opportunity to efficiently share libraries among multiple unikernels on disk. Libraries shared by multiple unikernels are stored on disk once per unikernel image (ELF file) using them, instead of being shared among them (disk deduplication). This last drawback is not related to KSM but rather to the current behavior of Spacer.

4 Architecture

This section outlines the architecture we developed to address the challenges highlighted in the previous section. We employ the term "libraries" to encompass both internal [36] (e.g., lib-scheduler, lib-boot, etc.) and external libraries (e.g., OpenSSL [59], musl [21], etc.), as well as applications (e.g., SQLite [16], Nginx [66], etc.).

Our architecture was developed with the Spacer Aligner tool [27] as its foundation, upon which additional components were integrated. The Spacer tool, publicly available on GitHub [23], aligns unikernels using a global map to position libraries at fixed absolute addresses. It also supports ASLR through the use of trampoline tables[2], enhancing memory sharing across multiple concurrent ASLR instances.

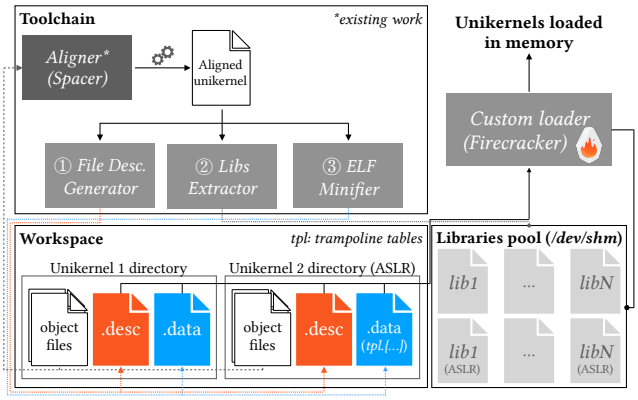


Figure 1: High-level architecture: Using the Spacer Aligner tool, our toolchain extracts libraries from unikernels into a shared pool (default: `/dev/shm`). A description file and application/library data enable the custom loader to run multiple unikernels by sharing common libraries.

The high-level architecture of our solution is depicted in Figure 1. Four major elements can be discerned in this diagram: a toolchain, a unikernel workspace, a pool of libraries and, finally, a custom loader (based on Firecracker [1]). As for Spacer, we have based our implementation on Unikraft [36] since it supports a wide range of libraries, is open source, and is under active development. Nevertheless, the complete toolset should work with minor changes for other library OSes and unikernels. In addition, the toolchain and the custom loader can handle both regular and ASLR Spacer unikernels.

4.1 Toolchain

The toolchain contains the essential tools for preparing unikernels, ensuring compatibility with our custom loader. The development of the toolchain involved integrating three new tools in addition to the Spacer tool (*Aligner*). This one generates aligned unikernels by using object files and custom linker scripts to place libraries at predefined memory addresses. Each tool can be executed *offline* (e.g., in a CI/CD pipeline). With our added contributions, the three distinct components are: ① *File Description Generator tool*: After aligning unikernels with the Spacer tool, a binary format description file (*.desc*) is generated for each unikernel by analyzing its underlying ELF file. This file includes a comprehensive list of libraries used by the unikernel, along with additional metadata such as whether the unikernel uses ASLR, and the start address and size of its data section. Each library entry contains the library name, its corresponding size, and its start address. We opted for the binary format to improve performance, but it can be easily converted to other formats such as YAML or JSON. ② *Libs Extractor tool*: As all

[1]The *Translation Lookaside Buffer* (TLB) is a page table cache that accelerates the translation of virtual addresses to physical ones.

[2]Tables that contain problematic instructions (e.g., those using addresses from other sections or libraries). There is one trampoline table per library.

unikernels are aligned, the code (`.text`) and the read-only data (`.rodata`) of different copies of a library are therefore identical. To avoid inflating unikernel size by including each library in every instance, this tool extracts (compiled) libraries as binary files into a dedicated library pool. We will discuss this pool further in Section 4.3. ③ *ELF Minifier tool*: Once all libraries are extracted, the underlying ELF file is stripped down to include only the aggregated data section (`.data`) of each library. All other sections and the ELF header itself are removed. This results in a minimal raw binary file that can be efficiently loaded directly into memory by the hypervisor. Note that if ASLR is used, the minimal binary file is slightly more complex and may contain several sections as illustrated in Figure 2.

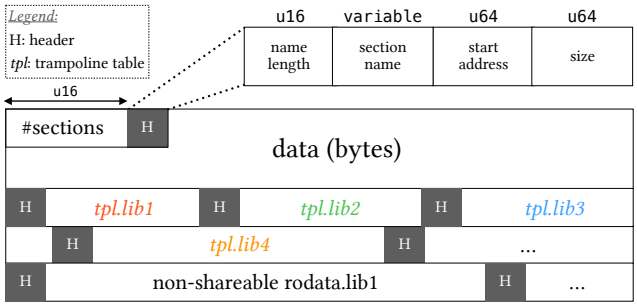


Figure 2: Representation of a minimal raw binary file used by the custom loader (ASLR only). In vanilla, only the data section (without its header) is included.

When using ASLR, a field representing the number of non-shareable sections is added at the beginning of the file. The file also contains the trampoline table (e.g., `tpl.lib{1-4}`) of each library. In this case, a header is added to the binary file, containing the library name (along with its length), the memory start address, and the size of each trampoline table. Still, for the ASLR configuration, some libraries may have their `.rodata` section not shareable. Indeed, some of them (e.g., libc and lwip) have relocation read-only data, which may vary depending on the order of the other libraries (since libraries are shuffled for ASLR configurations). These non-shareable sections are added to this minimal file instead of being extracted to the library pool. Note that these 3 tools can be executed in any order. However, a unikernel must be aligned by the Spacer tool before it can be processed by the rest of the toolchain.

4.2 Unikernels Workspace

This is the root directory containing a set of unikernels. Each unikernel is stored in its own folder. Before being processed by the toolchain, a folder contains the object files for a specific unikernel. Once the unikernel is processed by the toolchain, its directory contains a description file named *.desc* (specifying the libraries required by the unikernel) and a binary file named *.data*, which combines data, trampoline tables, and non-shareable read-only data (the latter two used only for ASLR) into a single and compact file that replaces the traditional ELF file.

4.3 Library Pool

Library code and their associated read-only data are extracted into an external library pool. By default, the *Libs Extractor tool* places these libraries in `/dev/shm`, which resides entirely in memory, providing significantly faster access compared with traditional disk-based storage. This approach not only accelerates boot times but also facilitates efficient sharing of libraries (in read-only mode) across multiple unikernels using a combination of `shm_open` and `mmap()`, as illustrated in Figure 3.

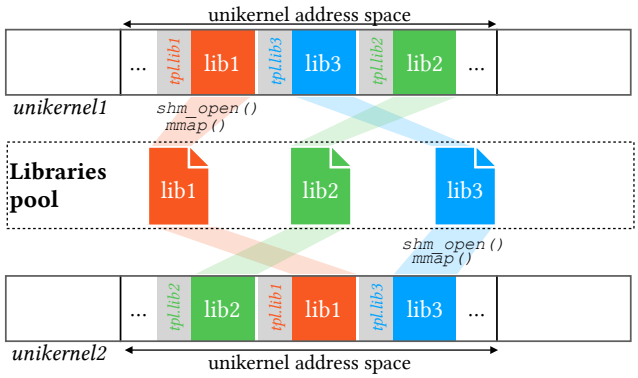


Figure 3: Aligned libraries are shared across unikernels via `shm_open()` and `mmap()` by the hypervisor, while trampoline tables remain unikernel-specific when using ASLR.

Each library contains two binary raw files: a vanilla version and an ASLR version (prefixed with the keyword `aslr`). This separation is necessary because the library code differs between the two versions. In the ASLR version, problematic instructions of each library are isolated in trampoline tables, unlike the vanilla version.

4.3.1 Managing the library pool. By default, all libraries are placed in `/dev/shm` by the *Libs Extractor tool*, based on the assumption that a unikernel will be deployed soon after being processed by the toolchain. Numerous essential libraries (e.g., ukboot, uksched, etc.) are required by various unikernel instances and are therefore preloaded into memory to reduce startup latency. This preloading provides a clear gain in startup performance, leveraging additional memory for efficient startup. The library placement behavior can be modified: the tool allows placing all libraries of a specific unikernel in its current workspace on disk instead (see Section 5.4). Regarding runtime management, the custom loader can be easily modified to monitor library usage and remove libraries from the memory pool once they become unnecessary, such as upon a unikernel's exit or through periodic cleanup (e.g., a cron job). This approach provides flexibility, allowing the system to trade off memory footprint against startup performance according to deployment requirements.

4.4 Custom Loader (hypervisor)

We developed a custom loader that uses this new unikernel representation to perform deduplication at load time, i.e. it directly maps the library pages onto the shared frames when loading the kernel image into main memory. Rather than developing a new loader from scratch, we modified the Firecracker hypervisor [22]. We chose

Firecracker instead of QEMU [9] because Firecracker has a much smaller code base and delivers better performance [1]. However, our implementation could be easily ported to other hypervisors.

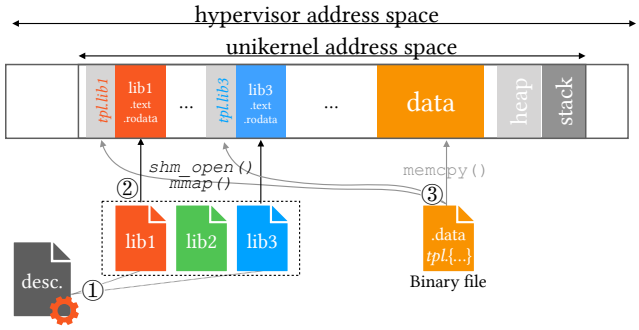


Figure 4: Our Firecracker-based loader parses a library description file (①), maps libraries to shared memory (②), and places non-shareable sections (e.g., data, trampolines) into the unikernel address space from a minimal binary (③).

Traditionally, the hypervisor allocates virtual memory for the unikernel using anonymous memory mapping. The size is determined beforehand, either specified in a configuration file or passed as a program argument. After being parsed (ELF parsing), the unikernel segments are loaded into this memory area and the unikernel is started by jumping to its start address. In our implementation, depicted in Figure 4, we depart from the conventional approach of providing an ELF file for the vanilla version of Firecracker. Instead, we supply a description file (containing a list of libraries along with their names, start addresses and sizes), alongside a minimal binary file containing library and application data. Subsequently, our loader parses the description file and iterates over the library list (step ①), mapping each library individually to shared memory via a combination of `shm_open()` and `mmap()` (step ②). In this case, the libraries are placed in shared memory segments, unlike the vanilla approach, where they were placed in anonymous memory. Moreover, complete parsing of the ELF file as well as the full unikernel extraction is no longer necessary since the libraries are already in memory. Only the data section and the trampoline tables are parsed and copied into the unikernel address space using `memcpy()` (step ③). Furthermore, our approach facilitates the management of unikernels using ASLR. Specifically, only the code and relocatable read-only data are mapped to shared memory segments, while trampoline tables and non-shareable read-only data are handled similarly to the data section. We implemented our custom loader on top of Firecracker v1.1.0; the corresponding patch is available on GitHub [24].

5 Evaluation

This section details the experiments conducted by comparing various unikernels and their configurations. Most of the following experiments involved at least four distinct configurations, including: (1) Default, which is the basic configuration used in Unikraft; (2) DCE, which involves activating Dead Code Elimination to minimize the size of unikernels; (3) Spacer, a method that enhances memory sharing by using libraries alignment; (4) Spacer-SLT (*Share at Load Time*), our custom loader based on Spacer. Additionally, we present the ASLR variants for certain experiments. To implement ASLR, we followed a similar approach to Spacer, which involves randomizing the memory layout during the linking stage (using linker scripts) [27]. In all ASLR setups, the libraries were rearranged, and a random offset was introduced between each library, creating varied images. All these operations were performed offline (before executing the unikernels).

Our evaluations covered short-lived and long-lived unikernels, deployed in heterogeneous (different unikernel applications such as web servers, databases, proxy servers, etc.) and homogeneous environments (several instances of the same unikernel). We aimed to simulate SaaS (Software as a Service) and FaaS (Function as a Service) environments by using unikernels with varying lifetimes and workloads. A comprehensive list of applications, including their functionalities and sources, can be found in our Github repository[3].

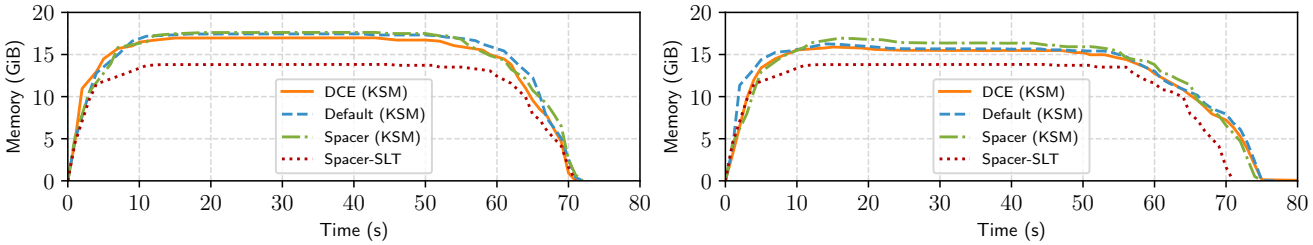
We sought to answer the following questions through our evaluation: (1) In Section 5.1: How effective is Spacer-SLT's deduplication compared with the conventional method of loading unikernels and relying on KSM to merge identical memory pages? (2) In Section 5.2: How consistent is Spacer-SLT's memory reduction compared with KSM under varying system load? (3) In Sections 5.3 and 5.4: What is the impact of managing libraries through our library pool mechanism? To answer this, we evaluated overall memory usage (unikernel + hypervisor), disk usage, and key performance metrics including page faults, TLB flushes, hypervisor boot time, and total execution time (covering creation, boot, run, shutdown, and destruction of the unikernel). We also measured the time required to load a library from disk when it is not cached. (4) In Section 5.5: What is the impact of running multiple ASLR-based unikernels? (5) In Section 5.6: What are the benefits of Spacer-SLT compared with a dynamically linked approach?

All experiments were conducted on a Debian 11 (bullseye) server using a Linux kernel 6.1, gcc 10.2.1 (GNU ld 2.35.2 as linker). Our current testbed with Linux kernel 6.1 prevented UKSM installation due to its discontinued maintenance. We discuss UKSM in more detail in Section 6.2. The host machine used for the experiments has 32 GB of RAM and an Intel Xeon CPU (E5-2650 v2 @ 2.60GHz) with 32 cores. In addition, Unikraft Enceladus 0.8.0 (with Firecracker support [80]) has been used for the experiments.

5.1 Memory Reduction: Spacer-SLT vs. KSM

This section evaluates the memory reduction gain offered by Spacer-SLT compared with KSM tuned according to 2 modes. In this experiment, a homogeneous environment was selected to focus on deduplication performance. Each KSM mode offers different trade-offs between convergence time and CPU overhead: (*i*) KSM-quiet: the default KSM setup that scans 100 pages every 20 ms, optimized for CPU usage but slower in merging identical pages into the same frame. (*ii*) KSM-full: A manually configured policy [68] that delivers memory efficiency for our workload, yet at the expense of considerable CPU consumption. For this setup, KSM is set to scan 20000 pages every millisecond. This setting uses the maximum values that continuously monopolize a full CPU core for KSM. We

[3] https://github.com/gaulthiergain/spacer-slt



(a) KSM-quiet: KSM takes time to identify and merge duplicate pages, resulting in minimal memory savings.

(b) KSM-full: KSM achieves better memory reduction by using more CPU cycles, but it is overwhelmed by the rapid arrival of many unikernels.

Figure 5: Evolution of the physical memory used by 128 identical FaaS unikernels (+hypervisor) being benchmarked according to 2 different configurations: (a) KSM-quiet and (b) KSM-full. Spacer-SLT achieves the lowest memory consumption in both cases without any performance penalties.

applied these two modes to a specific unikernel under the Default, DCE, and Spacer configurations to demonstrate the advantages of Spacer-SLT.

The benchmarked unikernels are implemented using the Go runtime, commonly used in FaaS environments. They behave as serverless functions intended for FaaS deployments. To emulate a bursty, high-density scenario, we sequentially launched 128 identical instances. Although typical FaaS functions often run for under one second [74], each instance here runs for approximately 10 seconds to observe the convergence behavior of KSM and avoid overstating its limitations on very short-lived workloads. During the experiment, we measured both memory usage and total execution time.

Figure 5a shows the initial performance of KSM with KSM-quiet. While KSM has a low CPU impact, it takes time to identify and merge duplicate pages, resulting in minimal memory savings. Spacer-SLT outperforms all other configurations in memory use. During the plateau phase, it consumes 24% less memory than DCE (KSM), 27% less than Default (KSM), and 30% less than Spacer (KSM). The difference in execution times between the configurations is negligible with KSM-quiet mode. Because this mode provided only minimal memory footprint reduction, we evaluated a more aggressive configuration (KSM-full), as shown in Figure 5b. KSM-full achieved better memory reduction overall than KSM-quiet, but the rapid arrival of many unikernels caused it to scan the memory of all instances without being able to merge pages effectively, leading to missed sharing opportunities. Spacer-SLT remained the superior performer, consuming on average 16% less memory than DCE (KSM), 19% less than Default (KSM), and 23% less than Spacer (KSM). In addition, this improvement in memory use came at a cost. We observed a slight increase in execution time for all KSM-based variants, taking 10% longer compared with KSM-quiet.

To further evaluate the efficiency of our approach, we measured the average physical memory usage per instance for the 128 FaaS unikernels from the previous experiment. The results, shown in Figure 6, highlight Spacer-SLT's ability to maintain consistent memory usage across all instances. In contrast, KSM-based configurations exhibit less stable behavior. With the KSM-full setting, the first few unikernels benefit from notable memory deduplication, as there are fewer pages to scan and merge. However, as more unikernels are launched in rapid succession, KSM becomes overwhelmed and unable to keep up with the growing memory footprint.

The KSM-quiet mode is overwhelmed from the outset due to its slow scanning rate, offering only marginal memory savings throughout the experiment. Even though we focused on Spacer with KSM, the same issue occurs with other configurations, such as DCE and Default (not shown for readability). In addition to average memory usage, we also illustrate the variation in memory consumption over time during each unikernel's 10-second lifetime. KSM-full exhibits greater fluctuations due to its policy, which can still merge portions of each unikernel's memory.

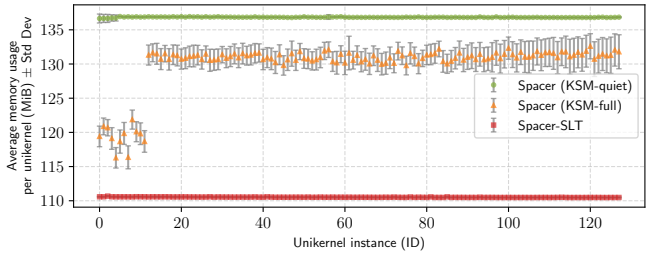


Figure 6: Average physical memory usage per instance for 128 benchmarked FaaS unikernels (including hypervisor). Spacer-SLT ensures consistent memory usage.

These experiments demonstrate that Spacer-SLT outperforms KSM in deduplication efficiency when handling a (large) batch of multiple unikernels running concurrently. KSM struggles to merge all sharing opportunities due to the high volume of memory pages it needs to scan and potentially merge.

5.2 KSM Variability vs. Spacer-SLT Consistency

To better understand how KSM's memory deduplication performance scales with system load, we measured the physical memory usage of a specific unikernel instance while varying the total number of concurrently running instances (4, 8, 16, 32, and 64). Each configuration was executed five times to capture variability, and results are shown in Figure 7. For readability, we only compared Spacer (with KSM-full) and Spacer-SLT. The plot reveals that, when

using an aggressive KSM mode, the memory usage of the observed instance increases with the number of concurrently running instances, while the variability across runs decreases. This trend suggests that KSM's ability to detect and merge duplicate pages is heavily influenced by system load. With fewer instances, KSM has more time and CPU resources to scan memory and perform deduplication, resulting in lower — but more variable — memory usage. As concurrency increases, KSM becomes overwhelmed by the growing memory footprint and cannot keep up with the rate of incoming pages, leading to higher and more stable (but less efficient) memory usage.

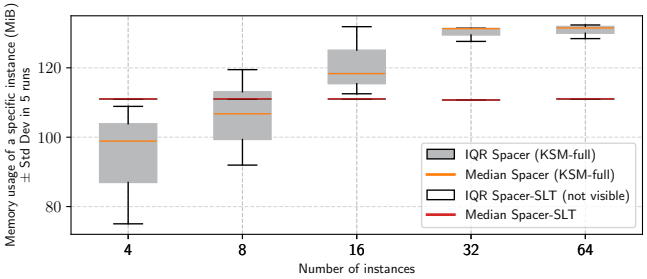


Figure 7: Memory usage of a specific single instance over 5 runs with concurrent unikernels. KSM shows increasing and less variable usage under heavy load. Spacer-SLT remains stable but slightly higher at low concurrency due to non-merging of writable pages.

In contrast, Spacer-SLT exhibits no variability across runs, consistently achieving the same memory footprint regardless of the number of running instances. However, unlike KSM, Spacer-SLT does not merge writable pages. As a result, in low-concurrency scenarios where KSM has time to deduplicate more aggressively, it can achieve better memory reduction. However, Spacer-SLT offers a predictable and efficient memory profile under load, making it more suitable for high-density FaaS deployments requiring scalability.

5.3 Evaluating Spacer-SLT

This section showcases Spacer-SLT's ability to achieve better metrics compared with the traditional approach of loading unikernels and relying on KSM for merging identical memory pages. To ensure a fair comparison, we evaluated Spacer-SLT against KSM with its default settings (KSM-quiet) because our testing criteria are more deteriorated by higher CPU usage than by slower KSM convergence speed.

We consider an environment that includes various applications ported as unikernels. We used 20 different applications [25] for some experiments: most of them have a network stack since unikernels are fairly well suited for a cloud environment. We then used different types of unikernels: short-lived unikernels that last only a few milliseconds and long-lived unikernels that last tens/hundreds of seconds. In addition, we also consider different workloads: some unikernels are idle and others are put under load. We limited ourselves to 20 applications because it would have been relatively time-consuming to port new ones.

First, we analyzed the size of the unikernel files (on the disk). Using Spacer caused inflation. This inflation is due to the header

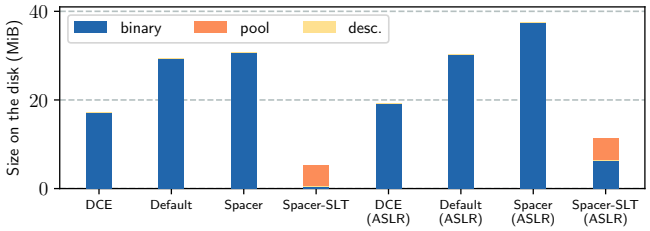


Figure 8: Spacer-SLT reduces disk usage for 20 unikernels by extracting common libraries into a shared pool.

string table section (.shstrtab) which contains more entries[4], and the trampoline tables for the ASLR version. Figure 8 illustrates the size occupied on the disk by the unikernel files. Since libraries (code and read-only data) are extracted into a common library pool, our approach considerably reduces the total size. For the vanilla version, it leads to a 3.3× reduction compared with DCE, a 5.5× reduction compared with Default, and 5.8× compared with Spacer (KSM). This reduction is smaller for ASLR versions where trampoline tables and non-shareable read-only data (different relocations) cannot be shared between instances, resulting in a 1.7× reduction compared with DCE unikernels, and in a 2.6× and 3.3× file size reduction compared with Default (KSM) and Spacer (KSM).

We then ran these 20 applications (one application per unikernel instance) to measure the total amount of memory used and to compare our approach with KSM. The total memory usage (for all unikernels + hypervisor) is depicted in Figure 9.

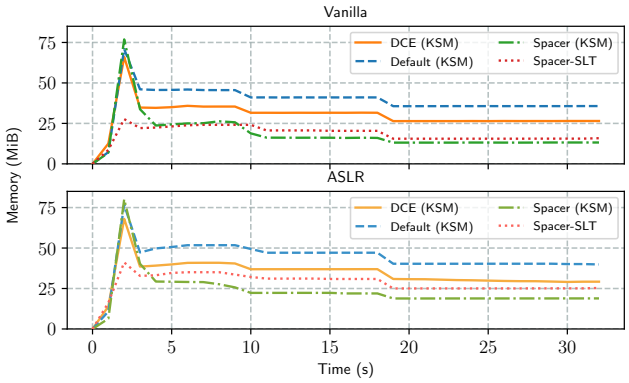


Figure 9: Evolution of the physical memory used by 20 different unikernels and the hypervisor (vanilla and ASLR) over time. Spacer-SLT performs memory deduplication at load time, which directly reduces memory usage.

With KSM, memory peaks appear at the beginning (from 0 to 4 seconds) and then slowly decrease thanks to page merging. With Spacer-SLT, the read-only pages are mapped directly to the same physical frames and memory peaks are reduced. It can also be seen that KSM can merge more pages than our custom loader. Indeed, after convergence, memory usage fluctuates between 13 MiB for Spacer (KSM) and 15.6 MiB for Spacer-SLT. This is because KSM

[4]Instead of having aggregated sections, they are disaggregated per libraries.

merges identical writable pages such as pages related to heap, video memory, etc. This happens mainly on idle unikernels that tend to have identical writable pages. For the ASLR version, the same observations can be made except that the consumed memory is a bit higher. This is due to the size of trampoline tables and non-shareable read-only data of some libraries (e.g., newlibc [65], lwip [71], etc.), which generate a memory overhead.

We also conducted controlled experiments to ensure that deduplication at load time does not impact the overall system performance. For Figures 10, 11 and 12, we isolated a single CPU core using `isolcpu` and ran a specific instance for measurement while pinning it to that core. In parallel, on the other cores, a real-world scenario is simulated by launching the 20 unikernels from the previous experiment. Among those, some are idle (like Nginx and DNS) and others are actively running (like zlib and SQLite). Eight unikernels were benchmarked, one at a time. We used the perf [28] tool to measure performance on the benchmarked instance, repeating the experiments 30 times for each of the 8 unikernels. Four were short-lived: a Hello World unikernel, a lambda function performing a 2× image upscale, a modified Nginx with full lwip support (network stack), and a modified DNS using partial lwip support. The latter two were stopped just before invoking `accept()` to simulate ephemeral unikernels. The other four were long-lived: a Mandelbrot generator that saves the result in an image file (1280 × 720 with 10000 iterations), a parallel 2000 × 2000 matrix multiplier that also saves the result in a file, the SQLite speed test [79], and a zlib unikernel that inflates and deflates a 10MiB file. This selection covers a range of short-lived and long-lived workloads.

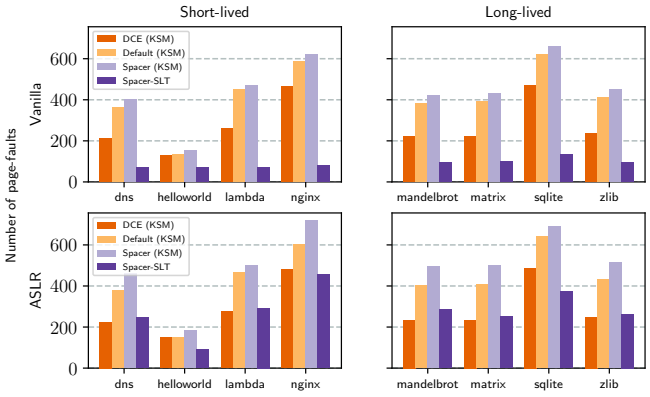


Figure 10: Page faults for vanilla and ASLR unikernels: Spacer-SLT reduces page faults in the vanilla configuration. However, under ASLR, the use of trampoline tables and non-shareable (read-only) data can lead to more page faults than with DCE.

To assess the library pool's effectiveness, we begin by analyzing the number of page faults. Figure 10 represents the number of page faults issued by different unikernel configurations (aggregated for the unikernel and hypervisor).

Our findings show that the number of page faults depends on the configuration. Spacer (KSM) generates the most page faults due to memory inflation. Compared with other configurations, Spacer-SLT significantly reduces page faults. For vanilla workloads, Spacer-SLT generates on average 3×, 4.6×, and 5× fewer page faults than DCE (KSM), Default (KSM), and Spacer (KSM), respectively. For some unikernels like Nginx, the reduction is even more substantial, with Spacer-SLT incurring 5.6×, 7×, and 7.6× fewer page faults than the configurations above. The low number of page faults in Spacer-SLT is likely due to its pre-loading of library code and read-only data into memory. For ASLR, the impact of Spacer-SLT is diminished because trampoline tables and non-shareable `.rodata` sections introduce additional page faults when loaded into memory. As a result, Spacer-SLT exhibits an average number of page faults similar to DCE (KSM), but it achieves 1.5× and 1.9× fewer page faults compared with the other two configurations.

We also analyze the hypervisor load time and the total execution time of different unikernels. In this context, load time refers to the time the hypervisor takes to initialize and fully load the unikernel into memory, without starting execution. Figure 11 presents these load times for different unikernel configurations. Although load time has only a minimal impact on long-lived unikernels, we include these results for completeness and consistency.

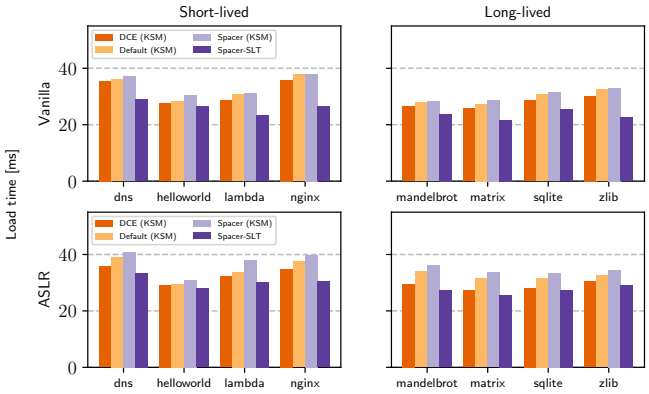


Figure 11: Load time (ms) for heterogeneous unikernels: Spacer-SLT is slightly faster than DCE, Default, and Spacer (KSM) as most libraries are pre-loaded into memory.

To measure load time, we used Firecracker's internal timestamping mechanism, which records the time elapsed until a specific I/O port is written [60]. As can be seen, Spacer-SLT reduces the load time since most of the libraries are already loaded into memory. Other configurations require parsing the underlying ELF file and then loading the unikernel into memory, which requires I/O and unnecessary copies to allocate frames (page faults). For the vanilla version, Spacer-SLT starts up faster than DCE (KSM), Default (KSM) and Spacer (KSM) by up to 1.2×, 1.3× and 1.4× respectively. As expected, the ASLR version incurs a load time penalty due to trampoline tables and non-shareable read-only data loading. In this case, Spacer-SLT with ASLR has on average similar load times to DCE (KSM) but remains faster than Default (KSM) and Spacer (KSM) by around 17% and 25%, respectively.

The total execution time is the time the unikernel takes to boot, run a certain load, shutdown and be destroyed. It is shown in Figure 12. Spacer (KSM) introduced a slight penalty in performance

due to memory fragmentation (i.e., biggest underlying ELF file to parse and more pages to load) [27]. Spacer-SLT changes the status quo and provides better results than its three counterparts.

This approach consistently yields lower execution times compared with KSM-based configurations. The first explanation for this difference is that KSM's memory scanning daemon consumes CPU cycles—especially in the case of long-lived unikernels, where KSM has enough time to analyze and merge identical pages. In contrast, short-lived unikernels might be partially missed by KSM. A second factor, which affects both short-lived and long-lived unikernels, is the overhead of fully parsing the ELF file and loading the unikernel into memory. This process involves I/O operations and redundant copying to allocate memory frames, which Spacer-SLT eliminates. On average, Spacer-SLT improves execution time by 9%, 11%, and 13% compared with DCE (KSM), Default (KSM), and Spacer (KSM), respectively. Although ASLR-based unikernels introduce a slight overhead, Spacer-SLT still demonstrates the same performance advantages.

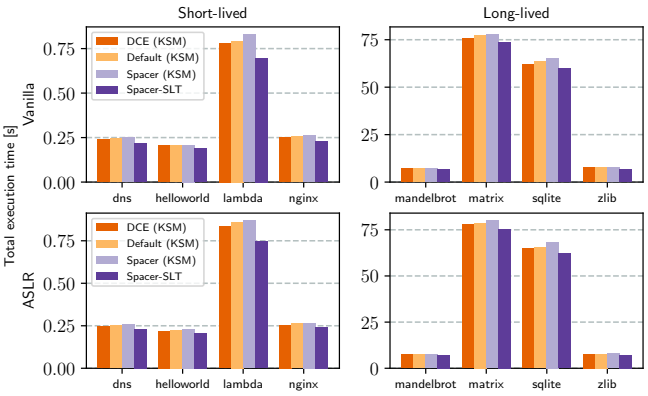


Figure 12: Total execution time (seconds) for different unikernel configurations: Spacer-SLT outperforms KSM-based configurations, especially for long-lived unikernels.

The number of TLB flushes of 20 different unikernels is shown in Figure 13. In this experiment, we employed ftrace [67] to analyze events associated with TLB flushes, enabling us to indirectly monitor TLB shootdowns by counting the number of "`flush_tlb*`" calls (e.g., `flush_tlb_all`, `flush_tlb_mm`, etc. [52]) invoked by KSM and the hypervisor. The benchmark is the same as the one used for Figure 9. We observe that the hypervisor produces fairly similar results between the different configurations, except for Spacer-SLT, which does much better since the libraries are already cached. When we look at the results with KSM, the number of TLB flushes is quite high, particularly for Spacer (KSM). Once again, this can be explained by the fact that more pages are merged into the same physical frame by KSM, invalidating the TLB and hence the higher number of TLB flushes. Spacer-SLT, which operates independently of KSM, significantly reduces TLB flushes compared with other configurations. It can achieve up to 10× fewer flushes compared with DCE (KSM), 12× fewer compared with Default (KSM), and 13× fewer compared with Spacer (KSM).

As demonstrated by our experiments, Spacer-SLT consistently outperforms KSM configurations, including the KSM-quiet setting.

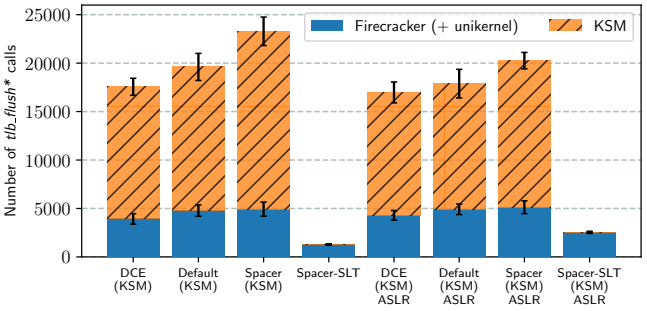


Figure 13: Number of TLB flushes issued by 20 heterogeneous unikernels (monitored at the hypervisor and KSM level) for vanilla and ASLR configurations. Spacer-SLT generates fewer TLB flushes than the KSM-based configurations.

The performance gap would be even wider when compared with a more aggressive KSM mode.

5.4 Managing The Library Pool

This section evaluates the impact of loading libraries when they are initially stored on disk. By default, our system preloads essential libraries into memory, assuming they will be used soon after unikernel startup. To quantify the cost of loading libraries from disk into `/dev/shm`, we measured the time our custom loader takes to move libraries from a dedicated unikernel workspace on disk into the shared memory pool, if the library is not already cached there.

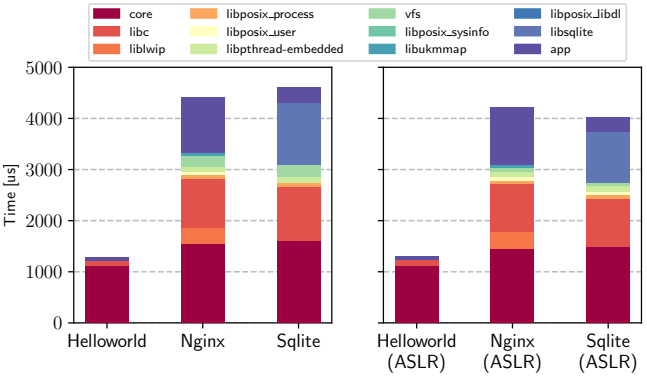


Figure 14: Our custom loader moves libraries to `/dev/shm`. Core libraries, libc, the application, (and libSQLite for SQLite) account for 80% of caching time.

For this experiment, we considered three different unikernels: Helloworld, SQLite, and Nginx (vanilla and ASLR). Figure 14 shows the time required for each library to be cached in `/dev/shm`. In general, copying libraries into `/dev/shm` has a minimal impact on performance. For instance, caching all libraries for Helloworld, Nginx, and SQLite takes only 1.3ms, 4.5ms, and 4.6ms, respectively. The significant difference between Helloworld and the others is due to Helloworld using a minimal libc, while the others rely on a larger libc. The core libraries, the libc and the application itself (as well

as libSQLite[5] for SQLite) take up most of the caching time (around 80%). For ASLR, the conclusions are similar, except for libraries with unshareable (read-only) data sections, which lead to smaller library sizes and faster caching. This applies to libc and libSQLite, which take less time for the ASLR version (e.g., 4ms for SQLite and 4.2ms for Nginx).

5.5 Running Multiple ASLR Instances

We also analyze multiple ASLR-based unikernels by running the same unikernel several times. This experiment uses a similar setup as Section 5.1 but with 64 ASLR-based unikernels. As before, we launched these instances sequentially, but this time, each with an underlying unique file (on disk) and memory layout[6]. The results can be seen in Figure 15.

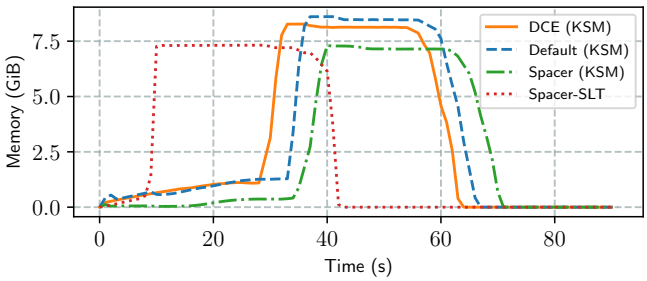


Figure 15: Memory usage of 64 FaaS ASLR unikernels (+hypervisor). KSM-based configurations exhibit delayed execution due to I/O wait times. Spacer-SLT reduces execution time by using a shared library pool as an I/O cache.

When using link time ASLR, each instance has a unique memory layout, enhancing security but introducing significant disk I/O overhead when instances load simultaneously, delaying execution. This overhead stems primarily from loading numerous distinct underlying files and is most pronounced when instances are launched in rapid succession. The effect is more observable with Spacer (KSM), which uses larger ELF files due to trampoline tables. Spacer-SLT mitigates this issue by caching libraries in memory, thereby reducing disk I/O to only essential components and achieving up to 1.7× faster execution than Spacer (KSM). When all unikernels are launched (at t = 9s for Spacer-SLT), Spacer-SLT exhibits slightly higher average physical memory usage than Spacer (KSM), as it does not merge writable pages. However, it still maintains lower memory consumption than the DCE and Default configurations. An in-depth discussion of the ASLR trade-off is provided in Section 6.4.

5.6 Spacer-SLT vs. Dynamic Libraries

As a final experiment, we compared Spacer-SLT with dynamically linked unikernels. Although Firecracker and Unikraft can be modified to support dynamic-linked unikernels, doing so would require substantial changes due to the position-dependent nature of certain KVM-internal libraries in Unikraft. To simplify development and reduce engineering overhead, we ported Spacer-SLT and the Unikraft codebase to userspace. We also extended Unikraft—originally limited to static linking—to support dynamically linked unikernels, which involved extensive modifications (50 files changed, 448 insertions, and 82 deletions), and implemented a lightweight userspace loader to run them. After these changes, we ported the same applications used in previous experiments, as well as a Python-based lambda function (with a 104MiB initrd), and systematically monitored memory usage and execution time, including that of the userspace loader.

Table 1: Total execution time (in seconds) of 7 heterogeneous unikernels (and the loader) across various categories.

Apps	Spacer-SLT	SLT [ASLR]	Dynamic
helloworld	2.11e-05	2.27e-05	3.00e-05
nginx	0.174	0.177	0.211
mandelbrot	0.358	0.359	0.370
lambda	0.539	0.550	0.555
python	2.114	2.137	2.271
zlib	3.814	3.853	4.095
sqlite-speed	47.503	47.857	48.529

Table 1 highlights the execution times of Spacer-SLT, Spacer-SLT (ASLR) and dynamically linked unikernels. The latter experiences a startup penalty as library relocations and symbol resolutions are performed at runtime (via lazy binding), impacting execution times, which Spacer-SLT avoids. As previously stated, applying ASLR to statically linked unikernels introduces a small overhead [27].

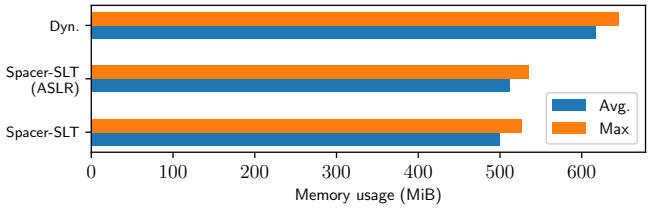


Figure 16: Memory usage of 7 heterogeneous unikernels (and the loader) across various categories. Spacer-SLT consumes less memory than dynamically-linked unikernels.

Figure 16 compares memory consumption (during 60 seconds) for the same unikernels running together, each with an infinite loop at the end of the workload. Dynamically linked unikernels consume, on average, 1.2× more memory due to the overhead introduced by the loader, dynamic linker (ld.so) and additional structures such as PLT/GOT tables. Once again, statically linked unikernels with ASLR show slightly higher memory usage than Spacer-SLT.

We discuss the relevance of using Spacer-SLT as opposed to dynamic libraries in more detail in Section 6.3.

5.7 Towards a Solution

Our evaluation demonstrates that the proposed architecture effectively addresses the issues outlined in Section 3. By using a custom

[5]For SQLite, we distinguish between the SQLite shell (application), and the library providing the SQLite API for interacting with the database.
[6]With our Unikraft implementation, ASLR is managed at link time, which creates a different underlying file per instance.

loader, we eliminate the need for a background deduplication daemon. Spacer-SLT performs memory deduplication at load time, thereby avoiding KSM's convergence time (§1). It also conserves CPU resources by eliminating the need to scan and merge memory pages across unikernels, leaving more processing power available for unikernels (§2). Additionally, Spacer-SLT provides consistent memory savings regardless of system load, unlike KSM, whose effectiveness deteriorates as concurrency increases, leading to degraded memory reduction (§3). The approach also reduces I/O overhead and TLB shootdowns, by caching and avoiding the allocation of redundant memory frames that would otherwise require merging (§4). By mapping only read-only pages, it avoids security risks and the performance overhead of deduplicating writable pages (§5). Finally, Spacer-SLT minimizes disk usage by disaggregating unikernels into shared libraries, eliminating the need for complete ELF binaries per instance (§6). Instead, each unikernel requires only a description file and a minimal binary containing non-shareable read-only data and trampoline tables (for the ASLR version).

6 Discussion

This section discusses various key aspects of our approach and implementation.

6.1 Combining Spacer-SLT with KSM

Spacer (KSM) uses less memory (after KSM convergence) than performing deduplication at load time. This is due to identical writable pages merged into the same single frame. Merging both read-only and read-write pages is the default behavior of KSM. Although this behavior allows consuming less memory[7] than Spacer-SLT, it leads to memory disclosure attacks which are based on a difference in write access times on the deduplicated memory pages that are recreated by CoW [3, 10, 76]. It also has several disadvantages as shown in our evaluation section. However, a cloud provider can always combine our approach with KSM by simply enabling it using our custom loader. The read-only pages will already be merged, and KSM will only merge the identical writable pages, resulting in lower CPU cycles. System administrators are expected to keep KSM on or turn it off according to their environments and type of workload, with all the consequences that it implies.

6.2 Using Another Deduplication Scanner

Improved memory scanners like UKSM [85] provide an enhanced version of KSM by prioritizing memory regions to accelerate the deduplication process. While UKSM offers improvements over traditional KSM configurations, it only applies through custom patches on Linux kernels 4 and 5. The project is also no longer actively maintained by its original developers [87], raising concerns about its long-term suitability for production environments. UKSM aims to balance the approaches of KSM-quiet and KSM-full configurations. While it merges more memory than KSM-quiet, it does not achieve the aggressive memory deduplication of KSM-full. This trade-off reflects UKSM's goal of optimizing memory deduplication efficiency without significantly compromising performance. However, UKSM still lags behind Spacer-SLT in both performance and memory deduplication. Spacer-SLT delivers the lowest execution time and memory footprint due to its proactive memory deduplication approach. Unlike traditional memory scanners, such as those in KSM and UKSM, which analyze all pages at runtime, Spacer-SLT performs this analysis during load time. By eliminating CPU-intensive runtime scanning and merging, and by leveraging caching, Spacer-SLT achieves superior performance.

[7] Especially when identical long-lived and inactive unikernels are running on the same server.

6.3 Using Spacer-SLT or Dynamic Libraries?

The primary goal of Spacer-SLT is to provide a simple and scalable system for performing memory deduplication at load time on statically linked binaries, effectively replacing KSM. It is not designed to support dynamic libraries. Unlike traditional dynamic libraries, which are typically confined to specific operating system environments, Spacer-SLT uses a lightweight, system-agnostic representation. This design facilitates adaptation to formats beyond ELF, including Windows' Portable Executable (PE) format, requiring only minimal modifications to the toolchain. It also allows multiple formats to be supported simultaneously. While supporting dynamic libraries could be a future direction, it introduces complexity, security issues and overhead due to dynamic loading, and symbol resolution. Moreover, as most unikernel projects use static linking, Spacer-SLT aligns naturally with their architecture and can leverage compiler optimizations, which are incompatible with dynamic libraries. In summary, dynamic libraries' support would require changes to the paradigm and architecture, adding runtime overhead and limiting adaptability across diverse systems.

6.4 Managing ASLR Efficiently

When thousands of instances are launched simultaneously, we have noticed that the use of ASLR has a significant impact on performance. This is a direct consequence of link time ASLR implementation [27], which generates distinct binary files for each instance and thus has an impact on I/O operations. In Section 5.5, we considered an extreme configuration in which all instances differ (libraries are also shuffled). This approach provides greater security but negatively impacts performance. Alternatively, some instances can be identical (sharing the same memory layout and underlying file), which improves performance. Another possibility is to assign one ASLR configuration per machine and change it periodically offline. Overall, these choices balance security and performance.

6.5 Spacer-SLT and Snapshots

Snapshotting is a common technique used in FaaS platforms to reduce cold-start latency and enable fast instantiation of functions by reusing pre-initialized runtime states [2, 5, 75]. These snapshots typically capture a function's memory image after initialization, allowing subsequent invocations to resume from this preloaded state rather than starting from scratch. While effective, this approach introduces significant storage and I/O overhead—especially for heavyweight runtimes like Python or Go—where snapshots can exceed several hundred megabytes. Moreover, snapshot reuse is often complicated by ASLR, which introduces variability in memory layouts and may require generating new snapshots per instance.

Spacer-SLT offers a fundamentally different approach by performing memory deduplication at load time, without capturing

runtime memory images. It operates directly on minimized ELF files (which are considerably smaller than full snapshot states) to identify and share identical memory regions across instances. Unlike snapshotting, Spacer-SLT is fully compatible with ASLR, preserving security while enabling memory sharing. It also avoids the merging of writable pages, thereby reducing the risk of side channels. As a result, Spacer-SLT delivers a scalable, efficient solution for simplified and faster FaaS deployment.

6.6 Securing the Libraries Pool

In Spacer-SLT, a library pool is maintained by extracting libraries (code and read-only data) into `/dev/shm`, allowing unikernels to efficiently share memory and reduce redundancy. To ensure the security of this setup, strict roles and privileges must be enforced by the cloud provider: libraries must be created and managed with restrictive permissions — read-only for authorized users and writable only by the administrator responsible for maintaining the pool — to prevent unauthorized access or modification. These measures help Spacer-SLT maintain a strong security posture while enabling efficient and secure memory sharing.

7 Related work

Memory deduplication. Given the limitations identified in our evaluation of the memory deduplication scanners [6, 53, 64, 82, 85], this section presents a comparison between our approach and content-based deduplication techniques. Disco [12] uses transparent page sharing with deduplicating CoW disks, while Satori [54] enhances Xen [8] by analyzing para-virtualized virtual disks to improve sharing. Transcendent Memory [47] introduces an API for sharing identical memory pages across VMs. Unlike these para-virtualization-dependent methods, our approach operates without such modifications and is tailored specifically for unikernels.

If we refer to memory deduplication related to unikernels, there are fewer research papers on the subject. Most frameworks focus on reducing the memory footprint of individual instances but overlook high memory usage when running hundreds/thousands in parallel. Solutions like KylinX [89] use pVM forks and inter-pVM communication APIs, while Nephele [44] enables memory and I/O cloning, both relying on Xen and heavy hypervisor modifications. In contrast, our approach performs deduplication at load time without relying on forks or cloning and is straightforward to implement. ORC [68] enables fine-grained binary object sharing with strong tenant isolation via a minimal trusted computing base (TCB), relying on hardware memory capabilities. While both ORC and Spacer-SLT optimize memory deduplication, ORC uses object-level deduplication in a shared address space, requiring custom compilers, loaders, and specialized hardware like CHERI/Morello [29, 84]. Spacer-SLT, in contrast, abstracts unikernels into a shared library pool, achieving deduplication without hardware dependencies. It ensures security within a shared read-only pool and operates on standard x86_64 architectures, offering greater flexibility with existing systems.

Unikernels. Several prior works have focused on optimizing various aspects of unikernel design and deployment. These include efforts to reduce application porting complexity [33, 36, 58], as well as techniques for minimizing startup latency [11, 49]. Spacer-SLT complements these approaches by enabling secure, memory-efficient unikernels, further reducing cold-start overheads through library caching and early-stage deduplication of read-only memory pages. Efforts such as uIO [55], CubicleOS [70], and FlexOS [40] introduce intra-unikernel isolation mechanisms using Intel Memory Protection Keys (MPK) [31] to confine faults within memory domains. However, these systems do not address memory efficiency in multi-tenant serverless environments, nor do they offer integration with content-based deduplication or ASLR-aware memory sharing strategies as provided by Spacer-SLT.

Serverless computing. Serverless computing [7, 14, 32, 51], particularly FaaS [45, 73], has become a prominent research focus, with efforts targeting reduced start-up latency and memory usage. Solutions like SEUSS [13] optimize execution by reusing computed results, SAND [2] enhances scalability through elastic provisioning and adaptive load balancing, and OFC [57] introduces in-memory caching for resource-efficient function invocations. Faasm [75] employs lightweight isolation to minimize virtualization costs, while Medes [72] reduces memory duplication in warm sandboxes by merging memory regions across function instances. Although these approaches focus on reducing startup latency and resource consumption, they are not designed with unikernels in mind and fail to address the drawbacks of using external memory scanning mechanisms. UaaF [78] links serverless functions with necessary libraries in unikernels, leveraging VMFUNC [31] for low-latency cross-sandbox invocations and a novel programming model to reduce memory and boot time. However, VMFUNC poses security risks by allowing writable memory sharing between unikernels [3, 10, 34, 76]. Our approach avoids these risks by deduplicating only read-only memory pages, preventing such attacks. Additionally, it supports ASLR, manages heterogeneous unikernels, and provides direct comparisons to DCE, offering broader flexibility and security enhancements compared with UaaF.

8 Conclusion

In this paper, we have designed, implemented, and evaluated Spacer-SLT, a loader for statically-linked unikernels that enables deterministic read-only memory deduplication at load time. Spacer-SLT has been integrated into Firecracker, demonstrating significant improvements across various metrics, including reduced memory usage, decreased disk space consumption, and enhanced startup and execution times. These gains are achieved through a combination of fewer disk operations, reduced page fault rates, and minimized TLB flushes, compared to traditional scanner-based memory deduplication methods or dynamically linked approaches.

Spacer-SLT contributes to improving the efficiency of cloud computing infrastructure and reducing its energy consumption for a given workload, particularly for microservices and short-lived workloads like lambda functions. Furthermore, Spacer-SLT is easily adaptable to other unikernel ecosystems and hypervisors.

Acknowledgments

This work is supported by the CyberExcellence project funded by the Walloon Region, under number 2110186, and the Feder CyberGalaxia project.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.

[3] Delwar Alam, Moniruz Zaman, Tanjila Farah, Rummana Rahman, and M. Shazzad Hosain. 2017. Study of the Dirty Copy on Write, a Linux Kernel memory allocation vulnerability. In *2017 International Conference on Consumer Electronics and Devices (ICCED)*. IEEE, New York, NY, USA, 40–45. https://doi.org/10.1109/ICCED.2017.8019988

[4] T.E. Anderson. 1992. The case for application-specific operating systems. In *Proceedings Third Workshop on Workstation Operating Systems*. IEEE Computer Society, Los Alamitos, CA, USA, 92,93,94. https://doi.org/10.1109/WWOS.1992.275682

[5] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS Made Fast Using Snapshot-Based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 730–746. https://doi.org/10.1145/3492321.3524270

[6] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Proceedings of the 2009 Linux Symposium*. The Linux Kernel Organization, Montréal, Canada, 19–28.

[7] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. *Serverless Computing: Current Trends and Open Problems*. Springer Singapore, Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.* 37, 5 (oct 2003), 164–177. https://doi.org/10.1145/1165389.945462

[9] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[10] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*. IEEE, New York, NY, USA, 987–1004. https://doi.org/10.1109/SP.2016.63

[11] Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'15)*. IEEE, New York, NY, USA, 250–257. https://doi.org/10.1109/cloudcom.2015.89

[12] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. 1997. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.* 15, 4 (nov 1997), 412–447. https://doi.org/10.1145/265924.265930

[13] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698

[14] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The rise of serverless computing. *Commun. ACM* 62, 12 (nov 2019), 44–54. https://doi.org/10.1145/3368454

[15] Christian Collberg, John H. Hartman, Sridivya Babu, and Sharath K. Udupa. 2005. Slinky: static linking reloaded. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 34.

[16] The SQLite Consortium. 2007. SQLite. https://www.sqlite.org, accessed 11/07/2022.

[17] Will Dietz and Vikram Adve. 2018. Software multiplexing: share your libraries and statically link them too. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 154 (Oct. 2018), 26 pages. https://doi.org/10.1145/3276524

[18] Ulrich Drepper. 2006. How To Write Shared Libraries. *Structure* 16 (2006), 2009.

[19] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: an operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.* 29, 5 (Dec. 1995), 251–266. https://doi.org/10.1145/224057.224076

[20] Izik Eidus et al. 2009. The Kernel SamePage Merging (Linux source code). https://github.com/torvalds/linux/blob/master/mm/ksm.c, accessed 11/05/2022.

[21] Rich Felker. 2011. musl libc. https://www.musl-libc.org, accessed 25/01/2021.

[22] firecracker microvm. 2018. Secure and fast microVMs for serverless computing. https://firecracker-microvm.github.io, accessed 11/05/2022.

[23] Gaulthier Gain. 2022. Spacer: a tool to align unikernels. https://github.com/gaulthiergain/spacer, accessed 24/10/2022.

[24] Gaulthier Gain. 2025. Custom Loader based on Firecracker (patches). https://github.com/gaulthiergain/firecracker_custom_loader, accessed 13/08/2025.

[25] Gaulthier Gain. 2025. Spacer-SLT: Load time memory deduplication for unikernels. https://github.com/gaulthiergain/spacer-slt, accessed 07/11/2025.

[26] Gaulthier Gain, Benoit Knott, and Laurent Mathy. 2025. Efficient Versioning for Unikernels . In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*. IEEE Computer Society, Los Alamitos, CA, USA, 296–307. https://doi.org/10.1109/CLOUD67622.2025.00038

[27] Gaulthier Gain, Cyril Soldani, Felipe Huici, and Laurent Mathy. 2022. Want More Unikernels? Inflate Them!. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 510–525. https://doi.org/10.1145/3542929.3563473

[28] Brendan Gregg. 2014. Performance analysis tools based on Linux perf_events (aka perf) and ftrace. https://github.com/brendangregg/perf-tools, accessed on 23/06/2025.

[29] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. 2023. The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System. *IEEE Micro* 43, 3 (2023), 50–57. https://doi.org/10.1109/MM.2023.3264676

[30] Daniel Gruss, David Bidner, and Stefan Mangard. 2015. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I* (Vienna, Austria). Springer-Verlag, Berlin, Heidelberg, 108–122. https://doi.org/10.1007/978-3-319-24174-6_6

[31] Intel. 2019. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation.

[32] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[33] Antti Kantee and Justin Cormack. 2014. Rump Kernels No OS? No Problem! *USENIX; login: magazine* 39, 5 (2014), 11.

[34] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. 2006. Comprehensively and Efficiently Protecting the Heap. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/1168857.1168884

[35] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv–Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Philadelphia, PA, 61–72. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

[36] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248

[37] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Xi'an, China) *(VEE '17)*. ACM, New York, NY, USA, 15–29. https://doi.org/10.1145/3050748.3050757

[38] Simon Kuenzer, Joao Martins, Mohamed Ahmed, and Felipe Huici. 2013. Towards Minimalistic, Virtualized Content Caches with Minicache. In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (Santa Barbara, California, USA) *(HotMiddlebox '13)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/2535828.2535832

[39] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: A Unikernel for Extreme Scale Computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers* (Kyoto, Japan) *(ROSS '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2931088.2931093

[40] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: towards flexible OS isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 467–482. https://doi.org/10.1145/3503222.3507759

[41] Hugo Lefeuvre, Gaulthier Gain, Daniel Dinca, Alexander Jung, Simon Kuenzer, Vlad-Andrei Bădoiu, Răzvan Deaconescu, Laurent Mathy, Costin Raiciu, Pierre Olivier, and Felipe Huici. 2021. Unikraft and the Coming of Age of Unikernels. *;login* 1 (12 July 2021), 1–8.

[42] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. 1996. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J.Sel. A. Commun.* 14, 7 (Sept. 1996), 1280–1297. https://doi.org/10.1109/49.536480

[43] Maxime Letemple, Gaulthier Gain, Sami Ben Mariem, Laurent Mathy, and Benoit Donnet. 2024. *v*TNT: Unikernels for Efficient and Flexible Internet Probing. In *2024 8th Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, New York, NY, USA, 1–4. https://doi.org/10.23919/TMA62044.2024.10559079

[44] Costin Lupu, Andrei Albiundefinedoru, Radu Nichita, Doru-Florin Blânzeanu, Mihai Pogonaru, Răzvan Deaconescu, and Costin Raiciu. 2023. Nephele: Extending Virtualization Environments for Cloning Unikernel-Based VMs. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 574–589. https://doi.org/10.1145/3552326.3587454

[45] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. 2017. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, New York, NY, USA, 162–169. https://doi.org/10.1109/CloudCom.2017.15

[46] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. *SIGARCH Comput. Archit. News* 41, 1 (mar 2013), 461–472. https://doi.org/10.1145/2490301.2451167

[47] Daniel J. Magenheimer, Chris Mason, Dave McCracken, Kurt Hackel, and Oracle Corp. 2009. Transcendent Memory and Linux. https://www.kernel.org/doc/ols/2009/ols2009-pages-191-200.pdf, accessed 17/06/2021.

[48] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233.

[49] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. 2013. Enabling Fast, Dynamic Network Processing with ClickOS. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (Hong Kong, China) *(HotSDN '13)*. Association for Computing Machinery, New York, NY, USA, 67–72. https://doi.org/10.1145/2491185.2491195

[50] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 459–473. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins

[51] Garrett McGrath and Paul R. Brenner. 2017. Serverless Computing: Design, Implementation, and Performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, New York, NY, USA, 405–410. https://doi.org/10.1109/ICDCSW.2017.36

[52] David S. Miller. 2024. Cache and TLB Flushing Under Linux. https://docs.kernel.org/core-api/cachetlb.html, accessed 30/05/2025.

[53] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. 2013. XLH: More Effective Memory Deduplication Scanners Through Cross-layer Hints. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 279–290. https://www.usenix.org/conference/atc13/technical-sessions/presentation/miller

[54] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. 2009. Satori: enlightened page sharing. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (San Diego, California) *(USENIX'09)*. USENIX Association, USA, 1.

[55] Masanori Misono, Peter Okelmann, Charalampos Mainas, and Pramod Bhatotia. 2024. uIO: Lightweight and Extensible Unikernels. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 580–599. https://doi.org/10.1145/3698038.3698518

[56] Felix Moebius, Tobias Pfandzelter, and David Bermbach. 2024. Are Unikernels Ready for Serverless on the Edge? . In *2024 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE Computer Society, Los Alamitos, CA, USA, 133–143. https://doi.org/10.1109/IC2E61754.2024.00022

[57] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244. https://doi.org/10.1145/3447786.3456239

[58] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 59–73. https://doi.org/10.1145/3313808.3313817

[59] Inc. OpenSSL Foundation. 1998. OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org, accessed 29/05/2024.

[60] Samuel Ortiz. 2019. Measuring Firecracker boot time. https://gist.github.com/sameo/0647d6aaa36e73e6b536b51c29db1ead, accessed 30/05/2025.

[61] Open Virtualization Alliance (OVA). 2024. Kernel-based Virtual Machine. https://www.redhat.com/en/topics/virtualization/what-is-KVM, accessed 25/11/2024.

[62] Rodney Owens and Weichao Wang. 2011. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines . In *IEEE International Performance Computing and Communications Conference*. IEEE Computer Society, Los Alamitos, CA, USA, 1–8. https://doi.org/10.1109/PCCC.2011.6108094

[63] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 291–304. https://doi.org/10.1145/1950365.1950399

[64] Wei Qiu, Marcin Copik, Yun Wang, Alexandru Calotoiu, and Torsten Hoefler. 2023. User-guided Page Merging for Memory Deduplication in Serverless Systems . In *2023 IEEE International Conference on Big Data (BigData)*. IEEE Computer Society, Los Alamitos, CA, USA, 159–169. https://doi.org/10.1109/BigData59044.2023.10386487

[65] RedHat. 2004. Newlib: a C library intended for use on embedded systems. https://sourceware.org/newlib/, accessed 12/12/2021.

[66] Will Reese. 2008. Nginx: The High-Performance Web Server and Reverse Proxy. *Linux J.* 2008, 173 (sep 2008).

[67] Steven Rostedt. 2008. ftrace - Function Tracer. https://www.kernel.org/doc/html/v5.0/trace/ftrace.html, accessed 29/05/2025.

[68] Vasily A. Sartakov, Lluís Vilanova, Munir Geden, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. 2023. ORC: Increasing Cloud Memory Density via Object Reuse with Capabilities. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 573–587. https://www.usenix.org/conference/osdi23/presentation/sartakov

[69] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 546–558. https://doi.org/10.1145/3445814.3446731

[70] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: a library OS with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 546–558. https://doi.org/10.1145/3445814.3446731

[71] Savannah. 2002. LwIP - A Lightweight TCP/IP stack. https://savannah.nongnu.org/projects/lwip/, accessed 12/12/2017.

[72] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory Deduplication for Serverless Computing with Medes. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 714–729. https://doi.org/10.1145/3492321.3524272

[73] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1063–1075. https://doi.org/10.1145/3352460.3358296

[74] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 14, 14 pages.

[75] Simon Shillaker and Peter Pietzuch. 2020. FAASM: lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 28, 15 pages.

[76] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. Memory Deduplication as a Threat to the Guest OS. In *Proceedings of the Fourth European Workshop on System Security* (Salzburg, Austria) *(EUROSEC '11)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/1972551.1972552

[77] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2013. Implementation of a Memory Disclosure Attack on Memory Deduplication of Virtual Machines. *IEICE Transactions on Fundamentals of Electronics Communications*

and Computer Sciences E96-A, 1 (2013), 215–224. QC 20170104.

[78] Bo Tan, Haikun Liu, Jia Rao, Xiaofei Liao, Hai Jin, and Yu Zhang. 2020. Towards Lightweight Serverless Computing via Unikernel as a Function. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*. IEEE, New York, NY, USA, 1–10. https://doi.org/10.1109/IWQoS49365.2020.9213020

[79] The SQLite Development Team. 2024. SQLite Speedtest. https://github.com/sqlite/sqlite/blob/3d24637325188c1ed9db46e5bb23ab5d747ad29f/test/speedtest1.c, accessed 29/05/2025.

[80] Andrei Topala. 2022. Unikraft with firecracker-mmio-0.8 support. https://github.com/Krechals/unikraft/tree/firecracker-mmio-0.8, accessed 11/05/2022.

[81] Unikernels. 2019. Unikernels - Rethinking Cloud Infrastructure. http://unikernel.org, accessed on 14/09/2023.

[82] Carl A. Waldspurger. 2003. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* 36, SI (dec 2003), 181–194. https://doi.org/10.1145/844128.844146

[83] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels as Processes. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 199–211. https://doi.org/10.1145/3267809.3267845

[84] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: revisiting RISC in an age of risk. *SIGARCH Comput. Archit. News* 42, 3 (jun 2014), 457–468. https://doi.org/10.1145/2678373.2665740

[85] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. 2018. UKSM: Swift Memory Deduplication via Hierarchical and Adaptive Memory Region Distilling. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 325–340. https://www.usenix.org/conference/fast18/presentation/xia

[86] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. 2013. Security implications of memory deduplication in a virtualized environment. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12. https://doi.org/10.1109/DSN.2013.6575349

[87] Lukasz Zarnowiecki. 2016. The Ultra Kernel Samepage Merging (UKSM). https://github.com/dolohow/uksm, accessed 19/05/2025.

[88] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. https://doi.org/10.1145/3477132.3483569

[89] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 173–186. https://www.usenix.org/conference/atc18/presentation/zhang-yiming