Efficient Versioning for Unikernels

Gaulthier Gain
Montefiore Institute
University of Liège
Liège, Belgium
gaulthier.gain@uliege.be

Benoît Knott
Montefiore Institute
University of Liège
Liège, Belgium
bknott@uliege.be

Laurent Mathy
Montefiore Institute
University of Liège
Liège, Belgium
laurent.mathy@uliege.be

***Abstract*—Unikernels are specialized, single-address-space operating systems (OSes) tailored to specific applications. They offer strong isolation, low memory/disk footprints, and fast startup times-making them well-suited for cloud and serverless computing. However, deploying many of them at scale in cloud environments introduces new challenges. In particular, managing library updates and versioning in statically linked unikernels is difficult due to their tightly coupled structure. Unlike dynamically linked binaries, statically linked unikernels lack built-in versioning mechanisms. Consequently, even minor library changes result in entirely new memory layouts, which can significantly increase memory consumption when multiple instances run concurrently.**

We present Spacer-$\triangle$, a framework that improves memory sharing across statically linked unikernels with different library versions. Spacer-$\triangle$ uses differential analysis and library alignment to enable page-level sharing via memory deduplication scanners or a custom loader backed by a shared library pool. Our evaluation with Unikraft shows that Spacer-$\triangle$ reduces memory consumption and boot overhead while maintaining compatibility across versions. The framework integrates into existing unikernel build pipelines with minimal changes and is released as open source.

***Index Terms*—Unikernels, versioning, statically-linked, library-os, virtualization, cloud-computing**

I. Introduction

Cloud services are increasingly built from composable, loosely coupled microservices, enabling independent development, testing, and deployment. In such architectures, a single client request can trigger a chain of dependent microservices. Serverless computing, or Function-as-a-Service (FaaS) [1], [2], is well suited to this model due to its event-driven execution, fine-grained billing, and elastic scalability, simplifying application management and enabling function chaining.

However, key challenges in serverless platforms include cold-start latency—the delay during the first invocation of a function as its execution environment is initialized—and memory overhead, particularly when running multiple applications concurrently. Both issues can significantly degrade application responsiveness, especially under bursty or latency-sensitive workloads.

Unikernels offer a promising solution to these challenges in serverless computing. Built on the concept of library OSes—where applications are linked with only the OS components they require [3]–[7]—unikernels are specialized, single-address-space operating systems tailored to specific applications. By compiling applications into minimal, self-contained images with only the necessary libraries and drivers, unikernels achieve fast boot times and exhibit extremely low memory and storage footprints. Their lightweight architecture makes them well-suited for serverless environments that demand rapid autoscaling and low-latency startup, often outperforming traditional virtual machines (VMs) and containers [8]–[13].

This efficiency has made unikernels increasingly attractive for cloud computing, where fast instantiation, low resource usage, and strong isolation are essential for cloud-native workloads. Some unikernel projects, such as Unikraft [12], have developed dedicated cloud platforms like KraftCloud [14]. Others—such as OSv [10], MirageOS [8], NanoVMs [15], and Rumprun [16]—offer support for deployment on mainstream cloud providers including AWS [17], Microsoft Azure [18], and Google Cloud [19].

Most unikernel projects follow a statically linked model [12], [16], [20], [21], embedding all dependencies directly into the binary at compile time. This approach offers ease of deployment, high performance and strong isolation but introduces new challenges in memory deduplication and library versioning—particularly in multi-tenant environments where different versions of the same application or dependency may coexist. Indeed, in practice, many third-party dependencies are infrequently updated by developers [22], [23], leading to a situation where multiple versions of the same library must be supported simultaneously on a given server. This creates challenges in memory management: when unikernels are statically compiled with slightly different versions of the same library, even minor updates can lead to diverging memory layouts. Changes such as added, modified, or removed functions result in unique memory pages across versions, thereby complicating deduplication efforts. Even with runtime memory deduplication scanners like Kernel Same-Page Merging (KSM) [24], these differences prevent the effective merging of different pages, leading to higher-than-expected memory consumption. Moreover, KSM's approach also introduces additional drawbacks, including high CPU overhead and/or slow convergence times [24]–[26]. Although load-time memory deduplication [27] using a custom loader can mitigate the runtime overhead of a memory scanner, it faces the same limitations in handling layout discrepancies, ultimately failing to achieve optimal memory efficiency.

Another challenge involves dynamically scaling new library updates and versioning. In traditional systems with dynamic

linking, multiple library versions can coexist, allowing developers to update or maintain applications independently of each other [28]. In static linking, all libraries are fixed at compile time, with each unikernel embedding its own static copy of the required libraries. Even minor changes between library versions can lead to significant variations in memory layouts, extending beyond the modified library code and affecting the entire unikernel structure. For example, if a unikernel using the first version of a library is already running and another unikernel with a slightly updated library version is started, the memory layout may differ significantly. Despite sharing much of the same codebase, these minor differences prevent efficient reuse of memory and both unikernels need to be recompiled to match a uniform memory layout. The static linking model, while beneficial for unikernel isolation and performance, inherently limits opportunities for memory deduplication across different versions in cloud environments.

To address these issues, we propose Spacer-Δ, an extension of Spacer [29]. Spacer-Δ is designed to improve memory efficiency across statically linked unikernels with varying library versions. It leverages library alignment and differential analysis to detect and align common libraries at the page level. This enables sharing of identical pages across unikernels, even when they are built with different versions of the same library. In addition, it allows new unikernel instances to be launched without recompiling existing ones while achieving optimal memory reduction. Spacer-Δ performs binary rewriting prior to unikernel execution, avoiding any runtime overhead. While our current implementation targets Unikraft [12], the approach is generic and could be applied to other unikernel code bases with minimal modifications.

Our main contributions are as follows: (1) We introduce a novel methodology based on page alignment and differential analysis that reduces memory consumption (regarding frame usage) when multiple unikernels with different library versions run on the same machine. (2) From this approach, we derive Spacer-Δ, a proof-of-concept toolset that integrates into existing build pipelines and enables the creation of unikernels that maintain retro-compatibility across library versions. (3) We provide a performance evaluation of Spacer-Δ, comparing our approach to other configurations, including DCE-optimized images. (4) We discuss the limitations of Spacer-Δ and identify some possible areas for improvement.

Spacer-Δ and its benchmarks are publicly available at: https://github.com/gaulthiergain/Spacer-delta.

II. Background

This section outlines the unikernel building and loading process, explores memory deduplication challenges, and introduces Spacer—a library alignment technique that improves memory deduplication efficiency in cloud environments.

A. Unikernels

There exist two main categories of unikernels [20]: Language-based and POSIX-compliant unikernels. (1) The former are associated with a particular programming language and require rewriting each application using the given language platform's API. In this category, we find MirageOS [8] (OCaml), HaLVM [30] (Haskell), Erlang on Xen [31], ClickOS [9] (Click Modular Router) and runtime.js [32]. (2) The second category of unikernels aims at maintaining POSIX compatibility with existing applications by providing a larger code base. As a result, this type of unikernel offers a more straightforward approach to migrate existing applications, as they only require compilation from source code. OSv [10], IncludeOS [11], HermiTux [21] and Unikraft [12] are examples of this second category.

For POSIX-compliant and statically linked unikernels, all necessary libraries and application code are compiled into a single executable image. When the unikernel needs to be loaded, the hypervisor allocates anonymous virtual memory and maps each segment of the unikernel's ELF file to the designated memory space with appropriate protections. Once loaded, the hypervisor interacts with the unikernel via the Kernel-based Virtual Machine (KVM) [33] interface to manage low-level hardware operations.

B. Memory Deduplication

Memory deduplication is a memory saving mechanism that consists of identifying identical pages and merging them into a single copy, improving efficiency and lowering costs in cloud environments. It can be performed at runtime using a background scanner or at load-time via a custom loader with a shared library pool. One common runtime method is Kernel Same-page Merging (KSM) [24], which maps identical pages to a single frame. KSM uses red-black trees to manage page states: a stable tree for merged pages and an unstable tree for potential candidates. Memory deduplication can also be performed at load-time using a custom loader that leverages `mmap` to map shared memory regions. This approach relies on a pool of libraries (which are in `/dev/shm`) extracted beforehand from unikernel binaries, enabling these libraries to be shared across multiple unikernels when new instances are starting [27].

C. Aligning libraries with Spacer

While unikernels offer excellent performance and efficiency, their specialized nature presents challenges in cloud environments hosting multiple instances. Specifically, variations in library sets cause (a) library shifts relative to page boundaries, and (b) address-specific instructions like `CALL` or `LEA` that differ between instances. These two limit the effectiveness of the memory deduplication process between unikernels.

To address these challenges, the Spacer [29] tool was developed. Spacer aligns libraries across unikernel instances by assigning each library to a page-aligned absolute address. It analyzes all unikernels in a workspace and generates custom linker scripts to relink each unikernel accordingly. When a library is unused in a given instance, Spacer inserts zero-filled pages to preserve alignment. To support security features like ASLR, Spacer employs trampoline tables that handle

problematic instructions (e.g., `CALL`, `LEA`) referencing other sections or libraries. These are patched via binary rewriting, allowing identical libraries to be mapped at different locations across instances.

While Spacer introduces slight per-instance memory overhead, it significantly increases page sharing and reduces total memory usage when running multiple unikernel instances concurrently on a server.

III. Problem Statement and Solution

This section highlights the challenges associated with library versioning in statically linked unikernels and presents step-by-step solutions to address them.

Unikernels leverage the concept of a library OSes [3]–[7], where operating system functionality is modularized into independent libraries. These libraries may exist in multiple versions, each introducing different functions and symbols. Without optimization, the choice of library versions can significantly affect memory consumption. For instance, consider two statically linked unikernel instances using various versions of the same library (e.g., uklibV@v1 and uklibV@v2). When these unikernels are loaded into memory, their .text sections—which store binary instructions—are distributed across multiple memory pages (each page is either depicted as a grey or a red box in Figure 1). A critical issue arises due to library misalignment. Adding a new single function, such as `f4`, in the second version of the library shifts all subsequent code. This misalignment propagates through memory, affecting not only the modified library but also other libraries within the unikernel. In addition, different cross-references (such as `CALL` instructions) also make memory pages differ between instances. This default behaviour is illustrated in Figure 1. Similar misalignment issues can also occur in other memory sections, such as .data, further exacerbating the problem.

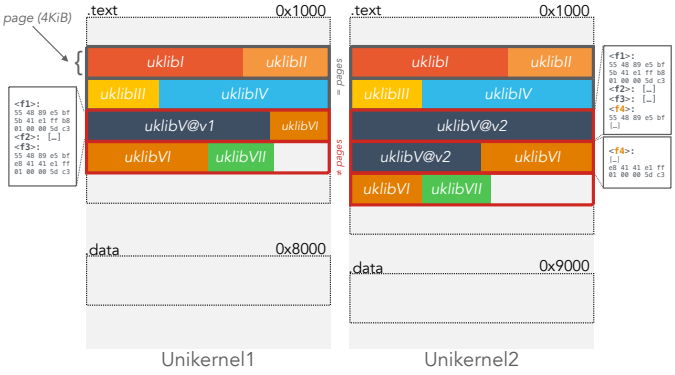

Fig. 1. Using multiple versions of the same library prevents memory deduplication across instances. Each page can store library code and is either shared (grey rectangle) or unshared (red rectangle).

Alignment (used by Spacer [29]) could be thought of as a potential solution. Aligning all libraries at the same fixed absolute addresses may ensure that identical pages are created across instances for the other libraries. These identical pages can then be efficiently shared using a memory deduplication mechanism, such as Kernel Same-page Merging (KSM) [24], or through a custom loader [27], significantly reducing memory usage. However, alignment becomes insufficient when handling various modifications between instances. For example, in the more complex and real scenario illustrated in Figure 2, uklibV@v2 introduces significant changes compared to its previous version. Functions are removed, their order is altered (e.g., new functions are inserted before existing ones), and some are modified. In such cases, alignment alone cannot resolve these differences, resulting in non-identical memory pages across instances and reducing the effectiveness of memory deduplication. Furthermore, if libraries call functions from the versioned library at different addresses, they will also remain unshared due to mismatched cross-references (not illustrated in the figures in this section). Although the diagram illustrates this issue with only three pages, the impact can scale significantly, involving hundreds or even thousands of pages if the library is larger and/or if multiple versions and instances are in use.

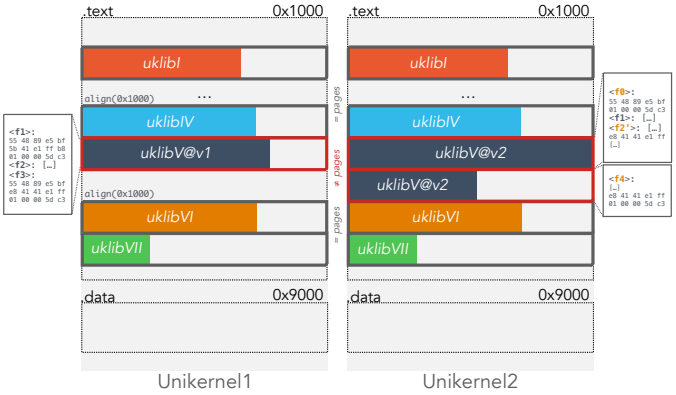

Fig. 2. Alignment becomes insufficient when handling various modifications between instances: functions may be added, removed, or altered. This results in differences in memory pages across instances and prevents effective memory sharing.

A simple approach could involve tracking the functions of a versioned library used across instances and organizing them in a specific order for each version, as done by Spacer for libraries [29]. Common functions would be placed at the beginning, followed by subsets of common functions, and finally, unique functions at the end of the library. Although promising, this approach faces challenges in achieving an optimal placement of functions when dealing with multiple instances and varying subsets of common functions. The presence of different subsets leads to significant complexity and numerous pages to manage across different instances. This concept is illustrated in Figure 3, where each function is assigned a fraction of a page. Functions are arranged based on their occurrences. For example, in the first unikernel, `f1` and `f5`, common to all three instances, are grouped onto the same page. Next, `f2`, present in two instances, is spread across two pages followed by `f4`. Finally, `f3`, unique to the first instance, is placed individually. With this approach, only two pages (green boxes) are identical and can be merged into a single

frame. The other pages are mapped to unique frames, requiring a total of 7 frames. Several variations of this method exist; some involve optimization techniques, such as bin-packing approaches [34]. However, these methods demand significant computational resources and become impractical when dealing with many versions and/or instances, making them unsuitable for library versioning. Another possible strategy is aligning each unique function to a separate page. However, this leads to significant internal fragmentation when functions are smaller than the page size (e.g., 4KB). Additionally, it increases the number of page faults, further reducing efficiency. Ultimately, none of these approaches effectively handles dynamic changes with instances, leaving the problem unresolved.

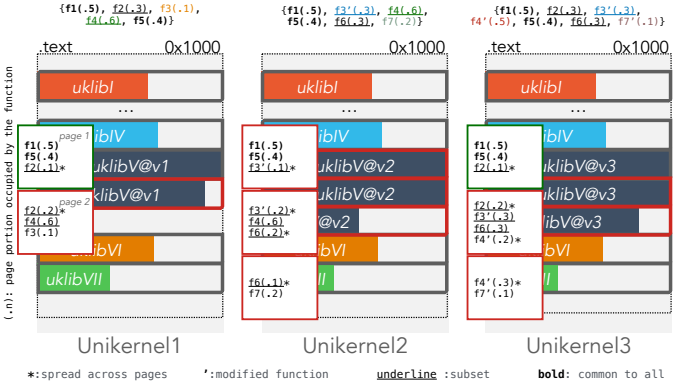

Fig. 3. Tracking all functions used across instances and arranging them in a specific order (per occurrence) for each version is not optimal. With this approach, only two pages (green boxes) are identical and can be merged into a single frame. The other pages (red boxes) are mapped to unique frames, requiring a total of 7 frames.

As previously mentioned, our goal is to maximize shared memory pages across instances while minimizing overall memory usage. Since the previously mentioned methods remain suboptimal, a new approach is required. To achieve this, we adopt a refined method that ensures backward compatibility between versions. In this strategy, called Spacer-Δ, each new library version is a delta of the previous version(s) reusing existing symbols and functions. The new version introduces modified variants of existing ones as well as new ones and places them at the beginning of a page. Figure 4 illustrates our method of preserving all pre-defined functions across instances by using the same library configuration as in the previous setup. For example, the second instance reuses functions defined in the first instance and introduces `f3'`, `f6` and `f7`, which are aligned to a new page. Although this approach increases the total number of pages, it significantly reduces the required memory frames due to memory deduplication. In this example, only 4 memory frames are necessary, offering substantial memory savings compared to the previous approach.

Additional steps are required to manage various library versions and maximize memory sharing. Placing dynamically versioned libraries may overlap with adjacent pages containing code or data, risking memory overwrites and unikernel crashes. Each new version or delta is positioned in the memory region

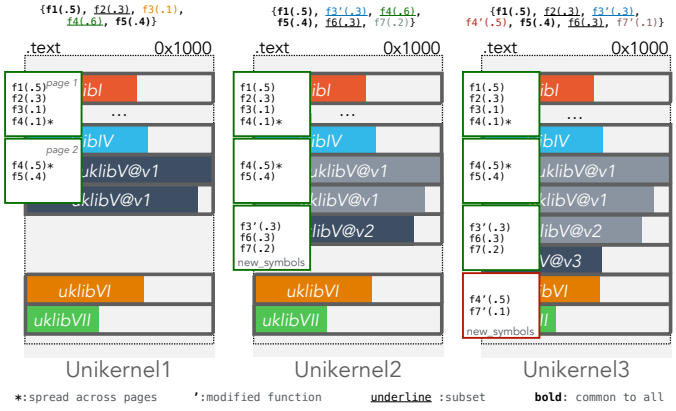

Fig. 4. Our approach leverages version backward compatibility by treating each new library version as a delta of the previous one. Existing functions remain fixed, while new functions are dynamically added to each instance, with the new library version aligned to page boundaries. Thanks to memory deduplication, only 4 frames are required.

between the heap and the stack to avoid these conflicts. Similarly, sections such as .data, .rodata (read-only data), and .bss (uninitialized data) are carefully organized to ensure no overlaps, maintaining memory integrity and stability. However, some configurations may still result in differences between instances. For example, if a function `f1` calls `f2` in one instance and then calls `f2'` in another, this creates a variation in the call instruction, making the corresponding memory pages non-shareable. To address this issue, trampoline tables (*.tpl*) are used [29]. These tables store the differing instructions (e.g., `CALL`, `JMP`, or `LEA`) that cannot be shared between instances. The original instruction is replaced with a `CALL` or `JMP` pointing to its new location within the trampoline tables [35]. Although these tables are not shared across instances, the library code pages remain shared.

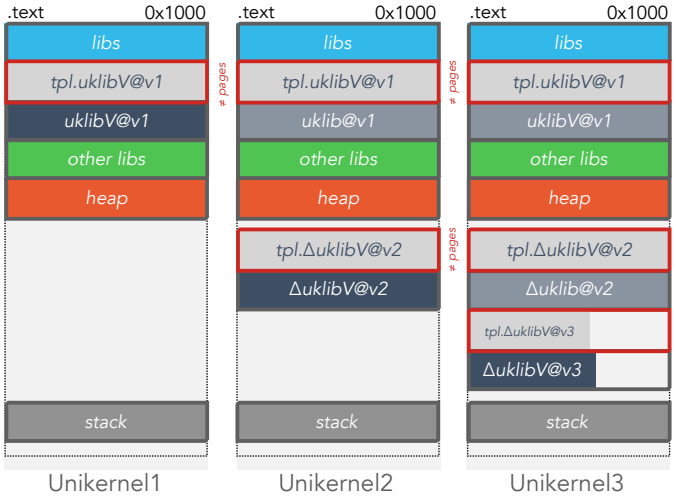

Fig. 5. The final representation of our approach to handling dynamically new instances and versions. Each new version (delta from the previous ones) is positioned between the heap and the stack. In addition, trampoline tables are used to isolate problematic instructions improving the sharing.

Additionally, trampoline tables are employed to handle

problematic instructions in other libraries referencing functions or data from the versioned library (or libraries). Figure 5 illustrates the final memory layout of this approach, with the trampoline tables for other libraries omitted for clarity.

IV. Architecture and Implementation

This section introduces the architecture designed to support versioning. Our implementation is based on Unikraft [12], [36], an open-source and actively maintained platform that supports a wide range of libraries. While our approach leverages Unikraft, the toolset can be seamlessly adapted to other library OSes and unikernels with minimal adjustments.

The versioning workflow, illustrated in Figure 6, consists of four distinct steps:

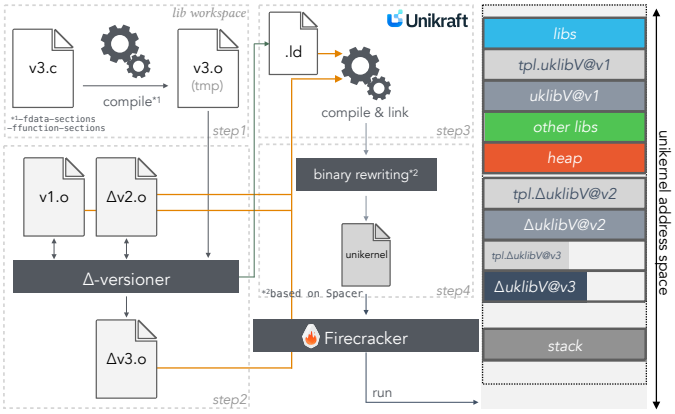

Fig. 6. Supporting versioning involves four key steps: (1) Object files compilation with specific flags, (2) Delta generation via a specific tool, (3) Unikernel linking, and (4) Binary rewriting to include the required trampoline tables. Finally, the unikernel is launched using the Firecracker hypervisor.

(1) Object files compilation: The library code must be first compiled with specific compiler flags (-fdata-sections and -ffunction-sections) to produce a temporary object file. These flags ensure that each function and data element in the source file is placed in a separate ELF section, essential for subsequent processing in our architecture. The underlying build system can be easily extended to include these flags [37]. *(2) Delta generation:* The temporary object file, along with all previous versions (also in object file format), is processed by a custom tool we developed called the Δ-versioner. This tool generates a new object file containing the delta relative to the previous library versions and can adjust symbols from previous object files, as detailed in Section IV-A. In addition, the Δ-versioner also produces a linker script specifying alignment rules for the libraries which is used in the next step. *(3) Unikernel linking:* The unikernel ELF file is generated by the linker, which combines the newly created object file with all other object files. This process is guided by a linker script [38], which specifies the locations of libraries and ensures proper integration. *(4) Binary rewriting:* The final step involves binary rewriting to isolate differing instructions into page-aligned trampoline tables. For this purpose, we leverage the binary rewriting tool developed by the Spacer authors [29]. The default behaviour of this tool was to replace instructions that reference addresses from other libraries into trampoline tables. We modify this behaviour to add trampoline tables for libraries that request services from libraries that are/could be versioned.

Once the processed binaries are ready, they can be seamlessly deployed to cloud platforms. Spacer-Δ maintains the standard ELF format, simply extending it with additional code and data sections to support versioning. This ensures full compatibility with existing hypervisors like Firecracker [39].

Spacer-Δ is also designed for effortless integration into existing software pipelines. The delta-processing phase—including binary alignment and rewriting—is performed entirely offline during the build process, introducing no runtime overhead. This offline approach makes Spacer-Δ particularly well-suited for automated CI/CD workflows.

Finally, the tool operates at the object file level and is compiler-agnostic[1], requiring no modifications to the compiler or linker. This design enables seamless integration with existing build systems. Further implementation details are provided in Section VI-A.

A. The Δ-versioner tool

As previously mentioned, backward compatibility with earlier versions is leveraged to dynamically support newer ones. To generate library deltas, we developed a custom tool called Δ-versioner, which processes ELF-format object files compiled with specific compiler flags. Introduced in Step 2 of Figure 6, this tool is written in C++ and relies on the ELFIO library [40] to parse and manipulate ELF files. It employs various data structures to efficiently manage symbols, sections, and relocation information.

The tool generates a final object file (which is a delta of the previous version) through the following steps: *(1) Parsing and processing object files:* It processes the input object files by extracting ELF metadata, including symbols, sections, and relocations. *(2) Mapping information:* For each function, the tool associates its corresponding symbol and relocations while identifying the content of the .data, .rodata, and .bss sections manipulated by the function. The use of specific compiler flags (-fdata-sections and -ffunction-sections) simplifies this process, as each function is placed in its own .text section with a corresponding .rela.text section containing its relocations. This contrasts with the original representation, where all functions and relocations are aggregated. *(3) Identifying new and modified functions:* The tool determines whether a function is new or modified by analyzing its symbol and name. It compares function sizes (from the symbol table) and, when multiple functions share the same name and size, uses hash values to compare their bodies. A function is marked as modified if its size or hash values differ. Additionally, if the content of any .data, .rodata, or .bss section manipulated by the function differs, the function is also considered modified. *(4) Adding unique elements to the new object file:* New and modified functions are added to the newly generated object

[1]Supposing -fdata-sections and -ffunction-sections (or equivalent) are available.

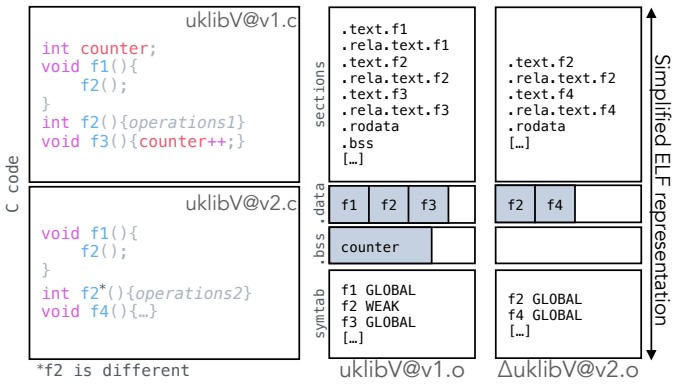

Fig. 7. Example of a simplified ELF representation of a lib versioned with our approach.

file through the following steps: *(4.a)* The symbol table is updated with the name of each new or modified function and its associated section. The symbol name is marked as strong (default value) [41] so that it can override a previous symbol of the same name during linking. Symbols with the same name from other objects are updated in the previous object files to be weak. *(4.b)* The content of the .data, .rodata, and .bss sections manipulated by the function (analysed via the relocations) is added to the corresponding sections in the resulting object file. *(4.c)* All the relocations associated with the function are updated and added to the object file. During this process, fields such as the addend and value [41] are recomputed to account for potential changes in offsets due to the addition/modification of content. We follow the same approach for handling relocations associated with the .rodata and .data sections. *(5) Object file generation*: When all previous operations are finished, the object file containing the delta of all previous functions is generated. Figure 7 provides a simplified representation of different versions. In this example, only new and/or modified functions (`f2`, `f4`) are added to the symbol and section tables. The corresponding .data and .bss sections are updated to reflect these changes. Relocations (not shown here) are also updated accordingly.

V. Evaluation

This section presents the experimental results, highlighting on memory savings achieved through our versioning approach, called Spacer-Δ. Experiments were conducted using Firecracker with KSM, or a custom loader based on Firecracker that performs memory deduplication at load-time. Although we rely on KSM for evaluation[2], our approach is agnostic to the underlying memory deduplication mechanism, as long as it supports page sharing. For KSM configurations, we used the default settings [43], which have a slight impact on performance [24].

We evaluated memory consumption, ELF file sizes, and application performance, using both traditional applications ported to unikernels and unikernels designed for lambda functions. We evaluated four configurations: (1) the Default Unikraft setup (without any optimization), (2) Dead Code Elimination (DCE) [44] to minimize unikernel size, and our versioning approach, Spacer-Δ, which has two variants: (3) Spacer-Δ using KSM and (4) Spacer-Δ (loader), which relies on a custom loader and a library pool for memory deduplication at load-time.

For certain experiments, we also present the ASLR variants. To implement ASLR, we adopted an approach similar to Spacer [29], which involves randomizing the memory layout during the linking stage using linker scripts [38]. In all ASLR configurations, the libraries were rearranged, and a random offset was introduced between each library, resulting in varied images. All these operations were performed offline, before executing the unikernels. Unlike vanilla versions, which include trampoline tables only for libraries that request services from versioned libraries, ASLR approaches require a dedicated trampoline table for each library. This results in an increased number and larger overall size of trampoline tables.

To manage different library versions, we rely on GitHub [45], where each library version is associated with a specific tag. The extent of changes between versions can vary widely depending on the developers' contributions. A new version might introduce substantial modifications, such as a completely restructured code architecture, or minimal adjustments, such as a simple bug fix in a function. For all libraries we tested, we cloned their GitHub repositories and checked some versions sequentially[3], in chronological order. We build a unikernel for the DCE and Default configurations for each version. Additionally, we employ the Δ-versioner tool to implement the approach detailed in Section IV for Spacer-Δ configurations. Table I represents the different libraries tested and their associated commits.

TABLE I
Library versioned and their respective commits (cloned from GitHub).

Library	commits (oldest to latest)
lib-sqlite [46]	6b54e32, fc44ea1, 2c6d801, 1da038f, 9927df2, 8dbe27e, d87000c, 60d9e2a
lib-pcre [47]	09fb6ce, 986d5c5, 1e0fcfc, 25c72e6, 2d6b260
lib-python [48]	a5f8ef1, 5900336, 04fad4f, dc93f53, 2d070a4,
lib-nginx [49]	0febe9a, 2eedb3f, 3229ec6, 6c5955f, 9cbe052
lib-pthread [50]	49a2433, 2dd7129, 955a702, bf7c1f6, e2705f9d

Our evaluation aimed to address the following research questions, with a particular focus on deployment in cloud environments: (1) Impact on memory and scalability (Section V-A): How effective is Spacer-Δ in reducing memory usage and improving scalability compared to conventional methods for versioning libraries in statically linked unikernels? (2) Impact on Disk Usage (Section V-B): What is the impact of Spacer-Δ on disk space consumption? (3) Impact on Performance (Section V-C): Does Spacer-Δ affect execution

[2]The default memory scanner in Linux kernel. Other scanners like UKSM [26], can provide better memory reduction but are obsolete [42].

[3]We skip versions that do not introduce any changes to the underlying codebase.

performance, and is it suitable for latency-sensitive cloud workloads?

To answer these questions, we assessed overall memory usage (unikernel + hypervisor), disk usage, and performance metrics such as total execution time. All experiments were conducted on a Debian 11 server running Linux kernel 6.1 with GCC 10.2.1 (GNU ld 2.43.50), equipped with 32 GB of RAM and an Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz (16 cores). We used Unikraft Enceladus 0.8.0 with Firecracker support [51] as the unikernel framework.

A. Memory usage

In the first experiment, we aimed to demonstrate that Spacer-Δ has lower memory consumption compared to the Default configuration and the DCE approach while providing dynamic scalability. To achieve this, we built Nginx unikernels, each using a different version of lib-sqlite [46]. We conducted the experiments using both the vanilla and ASLR-based versions. We aimed to demonstrate our experiment using a library that struck a balance between being neither too small (e.g., a single function) nor excessively large (e.g., comprehensive libraries like Python [48] or Go [52]). This choice allowed us to focus on a practical middle ground, ensuring meaningful observations without the overhead of extreme cases. Similar patterns can be observed across various versioned libraries, supporting our findings' generalizability.

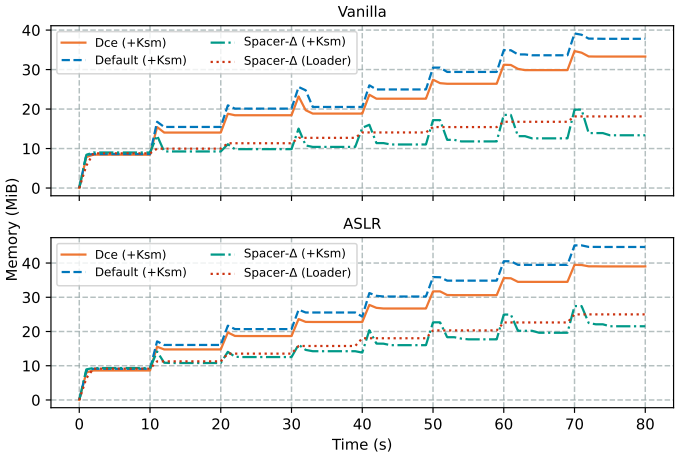

Fig. 8. 8 Nginx unikernel instances, each using a different version of lib-sqlite, are launched every 10 seconds. Compared to other configurations, Spacer-Δ can result in a reduction up to $2.8\times$ in memory consumption.

As shown in Figure 8, a new unikernel instance was launched every 10 seconds, starting at time 0, creating a dynamic scenario where new instances were continuously added while the previous ones remained active. For each KSM setup, a small memory peak appears whenever a new instance is launched, which is reduced as the memory scanner merges pages. For vanilla versions, Spacer-Δ (+KSM) can achieve significant memory savings, with a $2.5\times$ reduction compared to DCE and a $2.8\times$ reduction compared to the Default configuration. For Spacer-Δ (loader), memory pages are merged directly at load-time (via the custom loader and the pool of libraries), eliminating the peaks observed for KSM configurations. However, because this setup merges only read-only pages (code and read-only data) to mitigate Copy-On-Write (CoW) attacks [53]–[56], the overall memory consumption is slightly higher ($1.4\times$) than Spacer-Δ (+KSM).

For ASLR-based versions, similar observations apply, except that the average memory usage is higher across all configurations. This increase is attributed to reduced sharing opportunities. In the Default and DCE configurations, ASLR results in pages containing code being treated as different due to variations in library locations and resulting address differences. For Spacer-Δ, the higher memory usage stems from indirection tables and non-shareable (read-only) data, as library shuffling leads to different data/rodata relocations. Compared to DCE and Default, Spacer-Δ (+KSM) can respectively lead to a $1.8\times$ and a $2\times$ memory reduction.

B. Filesize

We then evaluated the impact of versioning on the unikernel size by analyzing the corresponding ELF files. This experiment was conducted using the same setup described in the previous section, with the disk usage of each configuration measured. Figure 9 illustrates the ELF sizes for the DCE, Default, and Spacer-Δ unikernels, as well as their respective ASLR variants.

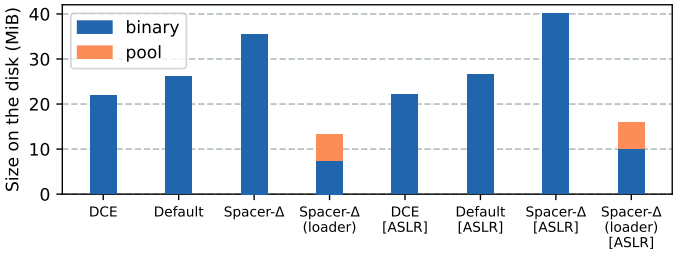

Fig. 9. The disk space occupied by 8 Nginx unikernel instances, each with a different version of SQLite, is significantly impacted by Spacer-Δ without a custom loader. However, integrating a custom loader and a shared library pool achieves optimal file size reduction.

The first observation indicates that using Spacer-Δ (+KSM) has a significant impact on file size. This increase is primarily due to the inclusion of all (previous) versions of a library within a single binary file, a design choice aimed for maximizing memory sharing. Consequently, this approach leads to substantial growth in ELF file sizes. Compared to DCE, which significantly prunes large monolithic libraries (e.g., libc), and the Default configuration, Spacer-Δ (+KSM) exhibits file size inflations of $1.6\times$ and $1.3\times$, respectively. Performing ASLR at link time introduces an additional increase in size across all KSM-based configurations. This is an artefact of the ASLR implementation, which relies on linker scripts. For ASLR setups, .text sections are no longer consolidated into a single entry but are divided into multiple entries, with one per library. As a result, the number of entries in the header string table increases, leading to a larger overall size. For the Spacer-Δ (ASLR) variant, this increase is further amplified by including

all trampoline tables, which are generated for each library. An important observation concerns the trade-off between file size and memory consumption. While DCE unikernels achieve notable disk space savings, this advantage is negated upon instantiation in memory, where Spacer-Δ variants exhibit reduced memory consumption, as illustrated in Figure 8.

Using a custom loader and a common library pool further optimizes file size reduction. By extracting common libraries (code and read-only data) into a shared pool[4], this approach significantly minimizes the total file size. For vanilla versions, this method achieves a $3\times$ reduction compared to DCE, a $3.6\times$ reduction compared to Default, and a $4.8\times$ reduction compared to Spacer-Δ (+KSM). For ASLR-enabled versions, however, the reduction is smaller due to trampoline tables and non-shareable read-only data caused by distinct relocations. Nevertheless, the custom loader approach still results in a $2.2\times$ reduction compared to DCE and $2.6\times$ and $3.9\times$ reductions compared to Default and Spacer-Δ (+KSM), respectively.

C. Performance

Finally, we conducted a series of performance tests to verify that our approach does not introduce performance degradation compared to existing methods. Given the complexity of performance testing—often influenced by factors such as memory allocators and workload characteristics—we focused on analyzing the total execution time, encompassing the create, boot, run, shutdown, and destroy stages of unikernels.

For the following experiments, we selected both short-lived and long-lived workloads, focusing on six applications, each with five different versions of a specific library. Three of the applications are short-lived: (1) A modified Nginx with full lwIP support (network stack), which is stopped just before the `accept()` function to simulate an ephemeral unikernel (using lib-nginx as the versioned library). (2) A PCRE-based unikernel that processes a 6MiB text file and returns matches for specific patterns (using lib-pcre as the versioned library). (3) A FaaS-based unikernel using Python that sorts a list of 2^6 elements and returns the result (using lib-python as the versioned library). The remaining three applications are long-lived: (4) A parallel 1500×1500 matrix multiplier that saves the result in a file (using lib-pthread as the versioned library). (5) The SQLite speed test [57] unikernel (using lib-sqlite as the versioned library). (6) The same Python FaaS function as before, but this time operating on a list containing 2^{16} elements. These three long-lived unikernels exhibit different memory usage behaviors. The SQLite test allocates a significant amount of memory, with a high degree of intra-sharing (i.e., self-sharing), primarily from heap allocations. In contrast, the matrix test shows minimal intra-sharing but still allocates some heap memory for matrix computations. The Python test has a larger codebase but allocates relatively less memory for list manipulation.

Two scenarios were considered for each unikernel: (1) a static approach, where each version is executed individually in standalone mode, and (2) a dynamic scenario, where new versions are continuously added while the previous ones remain active (modified to stay idle if their execution is short). The newly added instance is the one being benchmarked. We used the perf [58] tool to measure performance on the benchmarked instances, repeating the experiments 30 times. For Figures 10 and 11, we isolated a single CPU core using isolcpu and ran the benchmarked instance while pinning it to that core.

We begin by analyzing the first scenario in Figure 10, initially focusing on KSM-based configurations. For these configurations, we observe that Spacer-Δ (+KSM), with or without ASLR, introduces a slight execution-time overhead. This overhead arises because these unikernels must load additional pages (including trampoline tables) and because KSM needs to scan, and merge if applicable, a higher number of pages (for intra-unikernel memory deduplication).

In contrast, DCE unikernels achieve the fastest execution times, as they require fewer pages to load. For Default and DCE configurations, ASLR also has a minor performance impact due to the introduction of random offsets between different libraries. These offsets increase the overall size of the unikernel, leading to more memory pages that must be loaded, thus extending the loading time. Compared to DCE and Default, alignment optimizations do not improve execution time for Spacer-Δ (+KSM) and its ASLR variant.

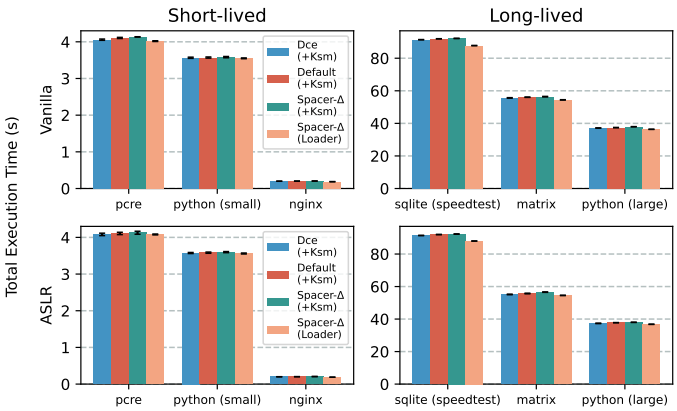

Fig. 10. Total execution time of several applications/versions using five different versions of a specific library, executed in standalone mode. Spacer-Δ configuration, combined with a custom loader, yields the best performance.

Using Spacer-Δ with a custom loader changes the status quo, yielding better performance than KSM-based configurations. This improvement is due to the library pool, which preloads the code and read-only data of libraries into memory. By reducing page faults, this approach minimizes delays associated with on-demand memory access. For ASLR-enabled unikernels, the impact of Spacer-Δ is slightly impacted. Trampoline tables and non-relocatable .rodata sections introduce additional page faults when loaded into memory. Nevertheless, the combination of Spacer-Δ and a custom loader consistently outperforms KSM-based configurations, as the scanning and merging process of the memory daemon consumes CPU

[4]A description file containing the list of libraries per unikernel is also provided to the custom loader to load only the required libraries.

cycles.

This performance advantage is particularly marked in long-lived unikernels, where KSM has sufficient time to analyze and merge identical pages. In contrast, short-lived unikernels experience minimal impact from KSM, as the time required for memory deduplication is too short to be entirely effective. However, even for short-lived unikernels, fully parsing the ELF file and loading the unikernel into memory—requiring I/O operations and unnecessary data copying to allocate frames—negatively affects execution time. For this experiment, Spacer-Δ (loader) can achieve execution time improvements of 7%, 9%, and 12% compared to DCE, Default, and Spacer-Δ (+KSM), respectively. Although ASLR-based unikernels introduce slight overhead, the same performance trends are observed for Spacer-Δ (loader), demonstrating its effectiveness even with ASLR.

In the dynamic scenario, the results illustrated in Figure 11 offer interesting insights. For long-lived unikernels, we observe greater variations in total execution time with KSM-based configurations. This is due to the increasing number of pages that need to be scanned and merged as more unikernels run concurrently. As additional instances are introduced, KSM cannot fully exploit merging opportunities due to its default configuration, which prioritizes CPU efficiency over optimal memory reduction. Consequently, the last instances experience less interference from KSM, resulting in a lower impact on unikernel performance. This is especially relevant for the SQLite speedtest, which often allocates memory for all its subtests, resulting in a large number of pages to scan. Since Spacer-Δ (loader) does not rely on KSM, there is an insignificant variation in the total execution time.

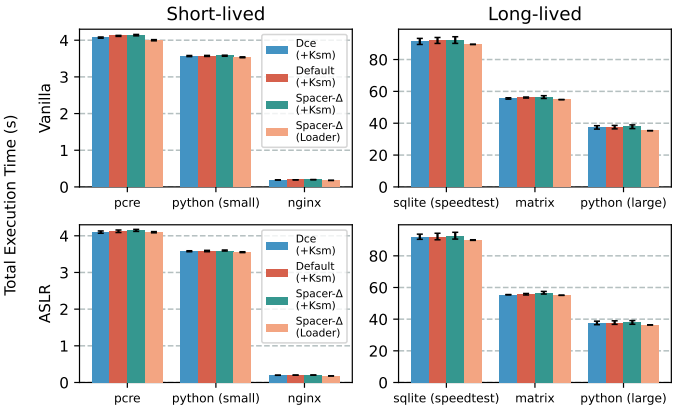

Fig. 11. Total execution time of several unikernels using five different versions of a specific library, executed with previous versions on different cores. Spacer-Δ (loader) loader, still yields the best performance.

For the final experiment, we aimed to analyze the effect of dynamically adding new benchmarked instances to the same core while keeping the previous ones active. We used the same setup as before, but instead of running instances on different cores, all instances were run on the same isolated core. We use the SQLite speed test unikernel (using lib-sqlite as the versioned library). Results are shown in Figure 12.

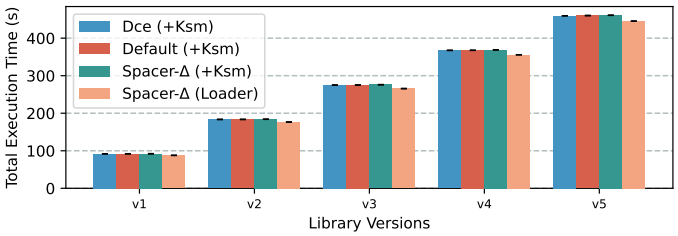

Fig. 12. Total execution time of five unikernels using different versions of lib-sqlite in the SQLite speed test. Spacer-Δ with a custom loader achieves the best performance.

In this scenario, the total execution time is significantly affected by the concurrent execution of multiple versions on the same CPU core. For all KSM-based configurations, even Spacer-Δ, no cache effect is observed on the performance since the underlying binaries for the different versions are different. As for the experiment depicted in Figure 11, there is also an increase in the number of concurrently running instances which leads to a proportional rise in the number of pages. However, because all instances are executed on the same core, the extended run-time allows KSM to merge and scan more pages, amplifying its impact and resulting in a performance penalty for configurations using KSM. This penalty is mitigated by using Spacer-Δ with a custom loader and a library pool, which yields a performance improvement of up to 6% compared to other configurations. This enhancement is primarily attributed to the library pool, which reduces I/O operations and avoids KSM overhead. The performance gap would be even wider when compared to a more aggressive KSM configuration.

Spacer-Δ enables optimal memory sharing, leading to improved memory reduction. When combined with a custom loader that performs memory deduplication at load time, it can further minimize filesize and enhance performance.

VI. Discussion

This section discusses various technical aspects of our implementation and explores potential adaptations to existing systems.

A. Integration and orchestration

Our system operates directly on object files, offering a compiler-agnostic design—assuming support for flags such as -fdata-sections and -ffunction-sections or their equivalents. This approach avoids any need to heavily modify existing toolchains, making it well-suited for integration into cloud development workflows, including CI/CD pipelines and build environments.

While versioning could alternatively be implemented at the compiler level—allowing tighter integration by embedding deltas directly into the ELF during compilation—such approaches typically require custom toolchains or plugins. Similarly, integrating binary rewriting into the linking or compilation stages may reduce processing steps, but again at the cost of modifying existing toolchains. These requirements

can hinder widespread adoption, as they often necessitate ecosystem modifications. In contrast, by operating at the object file level, our system remains lightweight and broadly flexible.

B. Dynamic Library versioning

While dynamic linking provides transparent memory sharing, it is generally less suited and less commonly used in unikernel environments. Unikernels are statically linked by design to ensure a minimal footprint, strong isolation, and fast startup [12], [15], [16], [20], [21]. Although some unikernels support dynamic linking [15], [59] it introduces additional runtime complexity, overhead, and potential security risks. Furthermore, dynamic linking typically manages major versions by requiring separate copies of library files for each major version, which means that entire symbol sets and code are loaded independently for each version. This can increase memory usage when multiple major versions coexist. In contrast, our approach retains the benefits of static linking while enabling memory sharing across different library versions through fine-grained differential analysis and page-level alignment—avoiding full duplication and significantly reducing memory overhead. Nevertheless, Spacer-Δ has the potential to be extended to dynamic libraries.

VII. Related Work

This section provides an overview of related work in the fields of versioning, unikernels, memory deduplication, and dynamic software updates, with a particular emphasis on their applications in cloud computing environments. It also outlines how these existing approaches differ from our proposed methodology.

Several unikernel frameworks, such as HermitCore [60] and Unikraft [12], have explored static linking optimizations to reduce cold start latency and resource footprints—key requirements for cloud-native systems. However, they do not offer systematic solutions for handling library versioning or library deduplication across multiple instances. Techniques like Spacer [29] improve memory efficiency in cloud scenarios by aligning library code across several unikernels to maximize sharing via memory deduplication. Nonetheless, Spacer focuses primarily on memory layout alignment and does not tackle broader challenges such as API evolution, backward compatibility, or multi-version coexistence on a same server. Solutions like KylinX [61] and Nephele [62] introduce mechanisms such as pVM forking and VM/unikernel cloning, aimed at reducing memory usage and improving deployment flexibility in cloud-based infrastructures. However, these rely on Xen [63] hypervisor modifications and still lack support for fine-grained versioning. While some unikernel projects [59] rely on dynamic loading, they do not address versioning in the context of statically linked unikernels. IncludeOS [11] offers "LiveUpdate" to enable low-downtime updates, yet it does not have a focus on memory optimization. Other projects, such as uIO [64]—which enables on-demand extensibility via a VirtIO-based file system interface—and SURE [65]—a unikernel-based serverless framework for fast, secure function startup—leverage the unikernel model but do not address fine-grained versioning.

Snapshot-based update approaches like SEUSS [66] and SAND [67] facilitate quick restores in cloud environments but do not handle ongoing versioning needs. Similarly, while tools such as Ksplice [68], MVEDSUA [69], and Jvolve [70] support live updates at the object or bytecode level, they are primarily designed for traditional operating systems and do not target unikernels or memory deduplication as core concerns.

VIII. Conclusion

Managing library updates and versioning in statically linked unikernels remains a significant challenge in cloud environments due to their compact structure and lack of built-in versioning mechanisms. Even minor library changes often result in entirely new memory layouts, leading to increased memory consumption when deploying multiple unikernel instances concurrently. To address these challenges, we introduced Spacer-Δ, a framework designed to optimize library versioning and memory efficiency in statically linked unikernels. By leveraging library alignment and a novel representation based on differential analysis between library versions, Spacer-Δ preserves memory layout consistency and enables efficient page-level sharing. Combined with the Kernel Same-Page Merging (KSM) and/or a custom loader, the framework effectively reduces memory overhead in dense deployment scenarios.

Our evaluation demonstrates that Spacer-Δ can achieve significant improvements in execution time, memory usage, and disk footprint, proving its practicality and efficiency. Its seamless integration with Unikraft and minimal adaptation needs for other unikernels make it a practical and versatile solution for cloud-native workloads. Spacer-Δ is available as an open-source project on GitHub, inviting further exploration and collaboration within the unikernel community.

IX. Future Work

We envision several future directions to extend Spacer-Δ: *(1) Handling additional languages*: Currently, our system is exclusively focused on the C programming language. We have conducted several tests to validate its functionality within this scope. Future work will expand support to include languages like C++, Go and Rust. Furthermore, we plan to explore and analyze the applicability of our approach to interpreted languages such as Python in subsequent studies. *(2) Snapshoting*: Versioning can be effectively combined with snapshotting. The general idea is to maintain a base snapshot of a specific unikernel on disk, which can be instantiated into different versions. This enables efficient state preservation and restoration for any version of the libraries within the snapshot. *(3) Dynamic Software Updating*: A last potential research direction includes developing mechanisms to dynamically patch the unikernel at runtime by applying deltas in the virtual memory. These efforts will pave the way for seamless and adaptive updates tailored to Spacer-Δ's architecture.

References

[1] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 405–410.

[2] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, p. 44–54, Nov. 2019.

[3] T. Anderson, "The case for application-specific operating systems," in *Proceedings Third Workshop on Workstation Operating Systems*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 1992, pp. 92,93,94.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: an operating system architecture for application-level resource management," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, p. 251–266, Dec. 1995.

[5] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," *IEEE J.Sel. A. Commun.*, vol. 14, no. 7, p. 1280–1297, Sep. 1996.

[6] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 291–304.

[7] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 195–211.

[8] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: library operating systems for the cloud," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, p. 461–472, Mar. 2013.

[9] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. USA: USENIX Association, 2014, p. 459–473.

[10] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "Osv: optimizing the operating system for virtual machines," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 61–72.

[11] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "Includeos: A minimal, resource efficient unikernel for cloud services," in *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, ser. CLOUDCOM '15. USA: IEEE Computer Society, 2015, p. 250–257.

[12] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici, "Unikraft: fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 376–394.

[13] F. Moebius, T. Pfandzelter, and D. Bermbach, "Are Unikernels Ready for Serverless on the Edge?" in *2024 IEEE International Conference on Cloud Engineering (IC2E)*. Los Alamitos, CA, USA: IEEE Computer Society, Sep. 2024, pp. 133–143.

[14] The Unikraft Authors, "Kraftcloud: True serverless," https://unikraft.cloud, accessed 19/05/2025.

[15] The NanoVMS Authors, "Nanovms," https://nanovms.com, accessed 19/05/2025.

[16] A. Kantee, "Flexible operating system internals: The design and implementation of the anykernel and rump kernels," Ph.D. dissertation, Aalto University, Finland, 2012.

[17] Amazon, "Amazon Web Services (AWS)," https://aws.amazon.com, accessed 19/05/2025.

[18] Microsoft, "Microsoft Azure," https://azure.microsoft.com/, accessed 19/05/2025.

[19] Google, "Google Cloud," https://cloud.google.com/, accessed 19/05/2025.

[20] Unikernels, "Unikernels - rethinking cloud infrastructure," http://unikernel.org, accessed 19/05/2025.

[21] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 59–73.

[22] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1513–1531.

[23] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Softw. Engg.*, vol. 23, no. 1, p. 384–417, Feb. 2018.

[24] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *Proceedings of the 2009 Linux Symposium*. Montréal, Canada: The Linux Kernel Organization, 2009, pp. 19–28.

[25] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "Xlh: more effective memory deduplication scanners through cross-layer hints," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. USA: USENIX Association, 2013, p. 279–290.

[26] N. Xia, C. Tian, Y. Luo, H. Liu, and X. Wang, "Uksm: swift memory deduplication via hierarchical and adaptive memory region distilling," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, ser. FAST'18. USA: USENIX Association, 2018, p. 325–339.

[27] G. Gain, "Spacer-slt," *unpublished*, 2024.

[28] U. Drepper, "How To Write Shared Libraries," *Structure*, vol. 16, p. 2009, 2006.

[29] G. Gain, C. Soldani, F. Huici, and L. Mathy, "Want more unikernels? inflate them!" in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 510–525.

[30] A. Wick and A. Chaudhry, "Cyberchaff: Halvm unikernels protecting corporate networks," http://unikernel.org/blog/2016/halvm-cyberchaff, accessed 19/05/2025.

[31] Cloudozer, "Erlang on xen," https://github.com/cloudozer/ling, accessed 19/05/2025.

[32] S. Iefremov, D. Björklund, and A. Abreu, "Javascript library operating system for the cloud." http://runtimejs.org/, accessed 19/05/2025.

[33] Open Virtualization Alliance (OVA), "Kernel-based virtual machine," https://www.redhat.com/en/topics/virtualization/what-is-KVM, accessed 19/05/2025.

[34] C. C. Lee and D. T. Lee, "A simple on-line bin-packing algorithm," *J. ACM*, vol. 32, no. 3, p. 562–572, Jul. 1985.

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

[36] The Unikraft Authors, "Unikraft," https://github.com/unikraft/unikraft/, accessed 19/05/2025.

[37] The GNU Project, "Gnu project - gnu coding standards," https://www.gnu.org/prep/standards/standards.txt, accessed 19/05/2025.

[38] The GNU Project (Free Software Foundation), "Binutils - gnu project - free software foundation," https://www.gnu.org/software/binutils/, accessed 19/05/2025.

[39] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434.

[40] S. Lamikhov, "Elfio - c++ library for reading and generating elf files," https://github.com/serge1/ELFIO.

[41] TIS Committee, *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (Version 1.2)*, TIS Committee, 1995.

[42] L. Zarnowiecki, "The ultra kernel samepage merging (uksm)," https://github.com/dolohow/uksm, accessed 19/05/2025.

[43] Red Hat, "Kernel same-page merging," https://docs.kernel.org/admin-guide/mm/ksm.html, accessed 19/05/2025.

[44] The GNU Project, "Optimize options (using the gnu compiler collection (gcc))," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, accessed 19/05/2025.

[45] GitHub, "Github," https://github.com, accessed 19/05/2025.

[46] The SQLite Consortium, "The sqlite amalgamation mirror with cmake," https://github.com/gaulthiergain/sqlite-amalgamation, accessed 19/05/2025.

[47] P. Hazel and Z. Herczeg, "Pcre - perl compatible regular expressions," https://github.com/gaulthiergain/lib-pcre, accessed 19/05/2025.

[48] The Unikraft Authors, "Unikraft port of python 3," https://github.com/unikraft/lib-python3/, accessed 19/05/2025.

[49] W. Reese, "Nginx: the high-performance web server and reverse proxy," *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[50] The Unikraft Authors, "Unikraft port of pthread-embedded, an embedded pthread library," https://github.com/unikraft/lib-pthread-embedded, accessed 19/05/2025.

[51] A. Topala, "Unikraft with firecracker-mmio-0.8 support." 2020, https://github.com/Krechals/unikraft/tree/firecracker-mmio-0.8, accessed 19/05/2025.

[52] The Go Authors, "The go programming language," https://go.dev, accessed 19/05/2025.

[53] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*. New York, NY, USA: IEEE, Aug. 2016, pp. 987–1004.

[54] D. Alam, M. Zaman, T. Farah, R. Rahman, and M. S. Hosain, "Study of the dirty copy on write, a linux kernel memory allocation vulnerability," in *2017 International Conference on Consumer Electronics and Devices (ICCED)*. New York, NY, USA: IEEE, 2017, pp. 40–45.

[55] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC '11. New York, NY, USA: Association for Computing Machinery, 2011.

[56] J. Xiao, Z. Xu, H. Huang, and H. Wang, "Security implications of memory deduplication in a virtualized environment," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[57] The SQLite Development Team, "Sqlite speedtest (speedtest1.c)," https://github.com/sqlite/sqlite/blob/master/test/speedtest1.c.

[58] B. Gregg, "The perf tool." https://github.com/brendangregg/perf-tools, accessed 19/05/2025.

[59] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 199–211.

[60] S. Lankes, S. Pickartz, and J. Breitbart, "Hermitcore: A unikernel for extreme scale computing," in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '16. New York, NY, USA: Association for Computing Machinery, 2016.

[61] Y. Zhang, J. Crowcroft, D. Li, C. Zhang, H. Li, Y. Wang, K. Yu, Y. Xiong, and G. Chen, "KylinX: A dynamic library operating system for simplified and efficient cloud virtualization," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 173–186.

[62] C. Lupu, A. Albiundefinedoru, R. Nichita, D.-F. Blânzeanu, M. Pogonaru, R. Deaconescu, and C. Raiciu, "Nephele: Extending virtualization environments for cloning unikernel-based vms," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 574–589.

[63] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, p. 164–177, oct 2003.

[64] M. Misono, P. Okelmann, C. Mainas, and P. Bhatotia, "uio: Lightweight and extensible unikernels," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, ser. SoCC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 580–599.

[65] F. Parola, S. Qi, A. B. Narappa, K. K. Ramakrishnan, and F. Risso, "Sure: Secure unikernels make serverless computing rapid and efficient," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, ser. SoCC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 668–688.

[66] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[67] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 923–935.

[68] J. Arnold and M. F. Kaashoek, "Ksplice: automatic rebootless kernel updates," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 187–198.

[69] L. Pina, A. Andronidis, M. Hicks, and C. Cadar, "Mvedsua: Higher availability dynamic software updates via multi-version execution," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 573–585.

[70] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: a vm-centric approach," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 1–12.