User-guided Page Merging for Memory Deduplication in Serverless Systems

Wei Qiu[1]*, Marcin Copik†, Yun Wang[2]‡, Alexandru Calotoiu†, Torsten Hoefler†
*Huawei Technologies, China
†Department of Computer Science, ETH Zürich, Zürich, Switzerland
‡Shanghai Jiao Tong University, China
†firstname.lastname@inf.ethz.ch

***Abstract*—Serverless computing is an emerging cloud paradigm that offers an elastic and scalable allocation of computing resources with pay-as-you-go billing. In the Function-as-a-Service (FaaS) programming model, applications comprise short-lived and stateless serverless functions executed in isolated containers or microVMs, which can quickly scale to thousands of instances and process terabytes of data. This flexibility comes at the cost of duplicated runtimes, libraries, and user data spread across many function instances, and cloud providers do not utilize this redundancy. The memory footprint of serverless forces removing idle containers to make space for new ones, which decreases performance through more cold starts and fewer data caching opportunities.**

We address this issue by proposing deduplicating memory pages of serverless workers with identical content, based on the content-based page-sharing concept of Linux Kernel Same-page Merging (KSM). We replace the background memory scanning process of KSM, as it is too slow to locate sharing candidates in short-lived functions. Instead, we design User-Guided Page Merging (UPM), a built-in Linux kernel module that leverages the `madvise` system call: we enable users to advise the kernel of memory areas that can be shared with others. We show that UPM reduces memory consumption by up to 55% on 16 concurrent containers executing a typical image recognition function, more than doubling the density for containers of the same function that can run on a system.

***Index Terms*—Serverless, Function-as-a-Service, Memory Deduplication, Inference**

Implementation: https://github.com/spcl/UPM

I. Introduction

Serverless computing allows users to deploy elastic and scalable applications decomposed into fine-grained and stateless functions, at the cost of increased overheads and duplicated resources. Deploying to FaaS allows developers to delegate operational concerns such as resource provisioning, maintenance, and fault tolerance to the cloud provider, and it has been used in parallel and compute-intensive tasks such as big data analytics, machine learning, and high-performance computing [1–5]. In FaaS, functions are deployed in isolated sandboxes for security reasons on dynamically allocated resources, and they can be scaled down to zero when traffic in the system decreases. While this operational model provides cloud operators with flexible and efficient allocation and scheduling, it leads to resource waste caused by duplicated function environments for each function instance [6–8].

Reducing memory pressure has many benefits for both the cloud operator and users. When function sandboxes can be co-located on the same host machine, resource utilization and data sharing through caches are increased [9, 10], and performance of data analytics is increased [11]. Exhausting available memory forces operators to remove idle and warm containers, increasing the rate of cold startups, a major issue in serverless computing [12]. Furthermore, reducing bloat increases the memory pool for low-latency storage needed in stateful serverless [13, 14]. Leaner platforms can host more functions with larger memory allocations, which allows serverless to process big data workloads faster and cheaper [1].

The landscape of memory consumption in serverless varies significantly, with 50% and 90% of different functions consuming at most 170 MB and 400 MB of memory, respectively [15]. A prevalent use case is machine learning inference [16–18], characterized by substantial computational requirements and large memory allocations that can require upwards of 500 MB of memory per instance [12, 19]. The trained model significantly contributes to this memory consumption, accounting for a major portion of process memory identical across functions (Sec. III). While deep learning inference has been moving to the edge, not all mobile and edge devices can support computationally intensive inference on large models [20, 21]. Serverless functions could be a perfect platform for implementing the required edge-cloud cooperation [22–24]. The growing interest in deploying machine learning in cloud functions, together with the increasing model size, requires new solutions to reduce memory redundancy and prevent resource exhaustion when scaling isolated and containerized function deployments.

Unfortunately, current memory sharing functionalities are insufficient for deduplication in serverless functions. Techniques such as Kernel Same-page Merging (KSM) are not designed to handle volatile and short-term workloads as they can require over half an hour to locate deduplication opportunities [25] (Sec. II). On the serverless side, systems adopt the ideas of caching and checkpointing function sandboxes [26–28], co-locating instances of the same function within a single process [6, 13], and decomposing containers into deduplicated state [7]. Such approaches are limited by sharing memory only

[1]Work on this paper was done while at ETH Zürich.
[2]Work on this paper was done while at Tencent Technology.

between instances of the same function, explicit programming of shared memory objects, and deep changes to the entire serverless software stack.

To address this issue, we propose **User-guided Page Merging (UPM)**, a built-in Linux kernel module that enables page sharing across different functions with an affordable time overhead. We extend the semantics of Kernel Same-page Merging by replacing the background memory scanning process with an on-demand, user-guided deduplication that better suits serverless workloads. We use the existing `madvise` Linux system call to implement the interaction between the user function and the deduplication running in kernel space. The *madvised* memory pages are treated as potentially mergeable regions. They are shared in a copy-on-write fashion if any page with identical content has been *madvised* before.

UPM is designed to be simple, flexible, low overhead, and easy to use. *UPM* is agnostic of language, runtime, and containers the FaaS system uses. In *UPM*, deduplication is applied only once on the cold container, and consecutive warm invocations benefit from page sharing without CPU overheads. Furthermore, *UPM* extends the security of memory sharing through KSM and page caches by introducing controlled, *opt-in* sharing. Using *UPM* is very simple and seamless, as users only have to annotate shareable memory regions, and manual data management with shared memory [13] is unnecessary. Our technique works for both file-backed and anonymous memory and enables sharing between different functions. We show experimentally that *UPM* successfully reduces the system's memory utilization by up to 55% when 16 concurrent containers are deployed.

We make the following contributions:

- We provide a thorough and detailed study on the memory sharing potential of serverless workloads (Sec. III).
- We propose *UPM*, a novel methodology for memory deduplication across serverless functions, and we implement *UPM* as a built-in module of the Linux kernel (Sec. V).
- We demonstrate empirically that *UPM* reduces up to 55% of memory consumption in a machine learning case study (Sec. VI).

II. Background and Related Work

Memory deduplication has been employed in datacenters for decades. However, such systems need several minutes to discover deduplication opportunities and are thus limited to long-running and stationary workloads. Serverless workloads require a different approach to support short-running invocations. While optimized serverless runtimes can provide memory sharing across invocations, they do not have the same level of generality offered by deduplication techniques.

A. Memory Deduplication

Page sharing has been proposed to share memory pages of files accessed by different virtual machines [29]. Content-based page sharing has been implemented in the Linux kernel as the Kernel Same-page Merging (KSM) [30]. There, memory pages are scanned and hashed to determine pages in guest virtual machines with the same content. When the hash value is already present in the system, a byte-by-byte comparison is conducted to guarantee that pages are truly identical and their contents have not changed. Then, pages are merged into one physical page in a copy-on-write manner, regardless of origin. An alternative approach requires modifying paravirtualized I/O to scan data read from disk to detect duplicated content in file-backed data and page caches [25]. The deduplication is most efficient on mixed CPU and I/O workloads, and it can reduce memory footprint by up to 60% [31–33].

a) Limitations: Memory pages are scanned periodically to keep the CPU overhead below 10-20% [34], and increasing deduplication efficiency requires tolerating higher CPU overheads [35], up to 68% when scanning every 20 milliseconds [36]. On the other hand, keeping the compute overhead moderate means that discovery of duplicated contents can take as long as 40 minutes [25, 37] and pages alive for less than 5 minutes cannot be deduplicated [25], as scanning just 50 megabytes requires over two minutes in the KSM [38]. In the serverless world, where the median and 90th percentile of function duration are about 3 and 60 seconds [15], respectively, functions will complete execution before the system finds sharing candidates of the function's memory. On AWS Lambda, function containers can be scaled down after only 6.5 minutes of no activity [12], which is not enough to discover deduplication opportunities. Furthermore, even if a function container is used frequently and retained long enough to benefit from page sharing, it would still handle many function invocations with bloated memory.

b) Optimized Sharing: Deduplication can be optimized with I/O-based hints [36] and page classification [39]. Still, the time needed for deduplication is in the order of minutes. Difference Engine [40] enables sub-page sharing, i.e., deduplicating pages that are not completely identical, by constructing patches containing the difference relative to a reference page. Such systems can outperform other deduplication techniques by locating pages with just a few bytes of difference that would otherwise never be shared. Similarly, virtual machines can be classified according to the similarity of memory contents or their software stack [33, 41], motivating co-location of machines recognized as similar.

c) ASLR: The address space layout randomization (ASLR) improves systems security, but it hurts the effectiveness of memory deduplication [42]. Sharing efficiency can be decreased by ASLR by 10-13 percentage points [43], and by 5% on FaaS workloads [7]. While these negative effects can be mitigated [44], the effects of ASLR on memory duplication in serverless are not widely studied.

The slow convergence of content-based deduplication was appropriate for virtual machines whose lifetime was weeks and months, but it does not fit into serverless containers.

B. Serverless

Function instances are placed in containers, microVMs, unikernels, and dedicated sandboxes. The sandbox allows a single server to handle many functions concurrently while preserving isolation guarantees. Unlike virtual machines requiring deduplication to remove identical files, containers share the page cache. With the help of virtual file systems such as Docker OverlayFS, the same files should have a single copy in memory across many containers.

To date, several systems have investigated how to reduce the memory consumption of serverless functions. SAND [26] and SOCK [27] use caching to reduce the startup time and share the runtime memory. They use the cache of previous instances for consecutive invocations of a function and then share the memory using copy-on-write, resulting in automatic memory sharing. Replayable Execution [28] and Catalyzer [45] adopt the Checkpoint/Restore (C/R) technique to reuse function memory. This technique checkpoints the application sandbox state as an image and consecutive invocations are restored from the checkpointed image. SEUSS [46] provides page sharing when deploying unikernels from a snapshot, and a similar approach can be used to pre-initialize functions at build time and deploy from an image heap [47]. Photons [6] and Faasm [13] share the runtime and application state by co-locating multiple instances of the same function within the same process. These systems focus on sharing the runtime and the application data across different invocations of the same function. None propose a general solution for memory sharing across *different* functions. Other techniques target memory sharing for specific applications, such as deep learning inference [48].

Medes deduplicates memory by introducing a new deduplicated state of serverless containers [7]. Warm containers are snapshotted, and their memory pages are fingerprinted to find duplicated content. While this method is independent of the function language, it requires deep changes and major additions in serverless platforms to support this new container state. Furthermore, it comes with new system components that need to be deployed on each server handling function requests.

Existing serverless runtimes can support memory sharing, but only in the limited case of invocations of the same function. Other approaches require dedicated runtimes and container systems. There is a need for an agnostic approach to memory deduplication that supports the rich serverless world of different languages, runtimes, and sandboxes.

III. Sharing Potential

The efficiency of memory deduplication varies depending on the system, employed workloads, and software stack similarity. For example, self-sharing within a single machine is often the primary source of memory savings and even minor software differences significantly impact inter-machine sharing [43]. On the other hand, deduplication on mobile operating systems is increased by 56% when considering sub-page sharing on 1kB segments [49]. Serverless functions represent a different class of workloads, as their memory

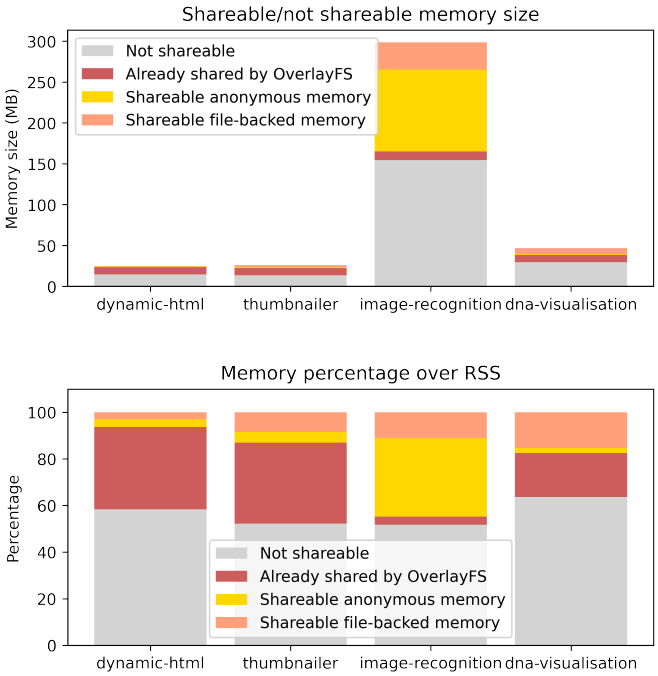


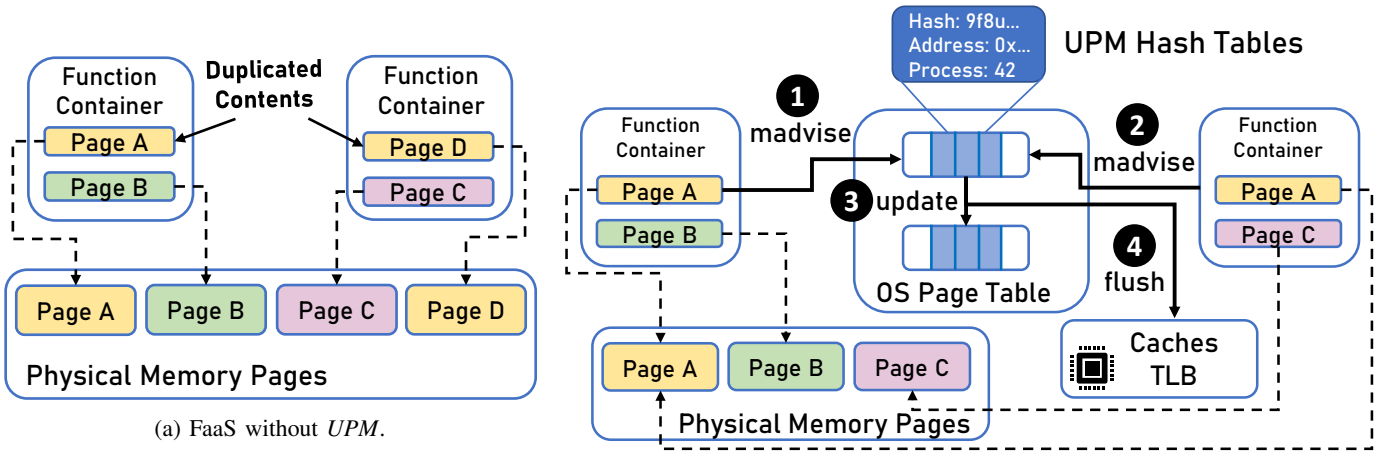
Fig. 1: Memory sharing potential in serverless functions.

footprint is dominated by language runtime and user code, and many operating system components are shared by default. Therefore, to understand how different deduplication strategies can perform on serverless workloads, we conduct a profiling study of functions to analyze the sharing potential, shareable memory type, sub-page sharing, and ASLR implications.

a) Setup: To investigate the sharing potential of functions, we profile the memory consumption of four different serverless functions from the SeBS benchmark suite [12]. We run functions locally on a virtual machine with the help of SeBS's local Docker runtime. The benchmark suite provides a Docker container with an HTTP server that accepts invocation requests and executes the function's code upon invocation. We measure the content similarity between different instances of the same function by running the function with changed inputs on two containers. The experiments are performed on an x86-64 virtual machine with 24 vCPUs Intel Xeon Platinum 8255C and 48GB of memory. We used Ubuntu 18.04 with kernel 4.15.18, Docker 20.10.6, Python 3.7.5, and SeBS 1.0.

For profiling, we select workloads representing diverse computation, memory, and I/O requirements:

- **dynamic-html** A lightweight web application that generates dynamic HTML from a template.
- **thumbnailer** A function that creates thumbnails of uploaded images with the help of the Pillow library.
- **image-recognition** A standard image recognition function, which downloads a pre-trained ResNet-50 model from storage and classifies a image using PyTorch.
- **dna-visualization** A function that generates a visualization for a given DNA data and stores result in the storage.



(a) FaaS without *UPM*.

(b) FaaS with *UPM*.

Fig. 2: **Memory deduplication with *UPM***: FaaS containers *madvise* duplicated memory regions to reduce duplicated memory.

Figure 1 presents the memory sharing potential of each benchmark. We include the volatile memory that is not identical, the file-backed memory already shared by OverlayFS, the identical but not shared anonymous memory, and the identical but not shared file-backed memory. For all of the benchmarks, the memory that is not shareable accounts for over half of the function memory. This includes the input, a large part of the memory footprint of small functions. Most of the file-backed library pages are shared by being page cache enabled in the Docker OverlayFS. A few libraries are loaded dynamically into memory, resulting in not much shareable file-backed memory. The ML inference function *image-recognition* shows the greatest memory-sharing potential compared to the others and the greatest overall memory footprint. There, about 40% of total memory can be shared, comprised of 27% and 13% in the anonymous and file-backed memory, respectively.

We notice that the memory of each benchmark grows soon after a request is received, then drops and remains constant once the request has finished processing. This phenomenon raises the question of whether memory deduplication should be applied at the moment when memory consumption reaches its peak. However, a closer inspection reveals that the request memory causes the increase and the magnitude of the increase is correlated with input data size. The sharing potential of input data size is minimal as identical inputs are unlikely.

b) Optimizations: To understand the potential for sub-page sharing, we select non-identical pages and verify if a sub-page of it would be shareable. We find that less than 2% of all pages are 75% identical, and less than 3% of all pages are 50% identical for each function. Thus, the overhead imposed by deduplicating data units smaller than a memory page would be much greater than the benefits of sharing.

Furthermore, we inspect the effects of address space layout randomization. By disabling ASLR, we increase the memory deduplication effect by 5.8 percentage points on average, which validates with another result achieved on a different set of functions [7]. Therefore, the ASLR is not a significant limitation for the effectiveness of memory deduplication.

c) Summary: To summarize, three of the four benchmark functions have limited deduplication potential because OverlayFS has already shared most of their identical memory through page cache sharing. On the other hand, the *image-recognition* function has significant memory sharing potential, with roughly 40% of memory that can be deduplicated. This result indicates the memory structure of machine learning inference functions makes them good candidates for memory deduplication. Finally, sub-page level sharing and disabling ASLR have little impact on the memory deduplication of serverless functions.

IV. User-guided Page Merging

We propose User-guided Page Merging (*UPM*), a new Linux kernel module for memory deduplication, named after the Kernel Same-page Merging (KSM). *UPM* supports sharing both anonymous and file-backed memory and is optimized to deduplicate the memory of serverless functions. *UPM* retains the classic ideas of hashing and copy-on-write page merging from KSM but replaces content-based page sharing with application hints (Figure 2). In this section, we outline the design goals and provide a high-level overview of the system, followed by a detailed analysis of *UPM* components (Sec. V).

A. Design Goals

UPM fulfills several requirements motivated by the characteristics of serverless workloads.

Speed Since content-based scanning is not fast enough to discover short-lived pages efficiently, *UPM* must be able to focus on identical pages in serverless environments instead of performing random scanning.

Focus on user data In virtual machines, 63-93% of shareable pages are part of the page cache [37]. Since page cache sharing is enabled by default in containerized workloads, *UPM* does not require specific treatment of I/O devices.

Stable Pages KSM is looking for stable pages that don't change their contents frequently. Similarly, *UPM* looks for

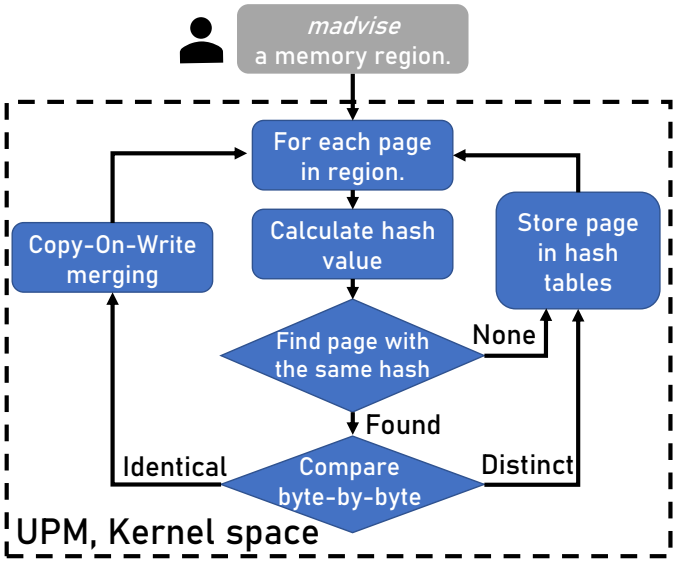


Fig. 3: The main deduplication algorithm of *UPM*.

memory pages that stay constant across invocations to avoid futile deduplication. Prior research on serverless functions indicates that at least 76% of memory pages are the same across invocations with changing inputs, and for a majority of functions, this ratio is larger than 97% [8]. Therefore, *UPM* can assume that consecutive invocations will not change most of the function's memory pages, and no additional classification of static pages is needed.

Compatibility The serverless landscape includes various runtimes and languages. Therefore, *UPM* should be compatible with existing systems, and should not require FaaS platforms to modify their internal implementation significantly. Since *UPM* is implemented as a kernel module, the deployment requires only an adjustment in the operating system configuration, leaving the FaaS runtime untouched.

B. *FaaS with* UPM

The overview of applying *UPM* to FaaS is presented in Figure 2. When a function begins executing in a container and initializes its data structures, it starts the page sharing process with the system call `madvise` (❶). The provided memory pages are scanned on the first container, and their hashes are inserted into *UPM* hash tables. When `madvise` is called from a second function container (❷), the call will detect pages with identical content. Then, the memory pages of the second container are replaced with references to pages of the first container with an update of OS resources (❸) and flushing hardware caches (❹). The deduplication takes place only on the first cold invocation in a container. For all consecutive executions, functions immediately benefit from the shared memory and incur no CPU overhead normally associated with memory deduplication.

C. *Guided Page Deduplication*

The distinguishing characteristic of *UPM* is that we delegate determining sharing candidates to the user. *UPM* does not scan the entire system memory. We believe users have adequate knowledge of application-specific data structures and can determine whether or not a memory region is a good candidate for sharing. The location of large and constant memory regions can be obtained with the help of profiling and analyzing container snapshots [8], a process that can be applied automatically to function invocations in the cloud. We let the application provide *hints*, a well-established technique previously used to optimize file system, networking, memory, and prefetching [50]. To that end, we use the existing *madvise* system call:

```
int madvise(
  void *addr, size_t length, int advice
);
```

This system call lets the kernel know how it is expected to handle a given memory area: `addr` is the virtual address in the memory, `length` is the size of the area, and `advice` indicates which kind of advice the program would like to give. Consequently, *UPM* can only share memory regions explicitly allocated by the user with known addresses.

The main idea of the algorithm is shown in Figure 3. When *UPM* is recommended to deduplicate a specific memory region, *UPM* calculates hash values for each page in the region and searches for previously reported pages with the same hash value. Then, a byte-by-byte comparison is applied to verify that pages are truly identical when the hash value matches. Afterward, the pages are merged in copy-on-write semantics. If any process attempts to modify the page's contents in the future, the operating system will replace the single copy of the deduplicated page with two instances.

UPM incurs no overhead unless users indicate deduplication candidates. Thus, *UPM* does not require kernel-level configuration and switching. It will remain inactive by default if no calls to the `madvise` system routine are made.

D. *Concurrency*

Unlike KSM, which scans and shares pages in the background using a single kernel thread, *UPM* shares pages synchronously, which means the function will be blocked by the `madvise` system call and will only continue to execute when the system call returns. Alternatively, the request can be processed asynchronously to avoid adding overhead to the function runtime. For example, deduplication can be executed in the background of the function invocation, or cloud providers can schedule this task between invocations.

UPM supports safe access to the pages being *madvised* and ensures consistency. Multiple containers can request *UPM* deduplication simultaneously. All accesses to data structures are protected with kernel spinlocks. We use write-protection and byte-to-byte comparison to ensure that the page being considered is not modified while the deduplication takes place.

E. *Security*

UPM extends the security model of KSM and shared page caches with *controlled* sharing. The hint generation is

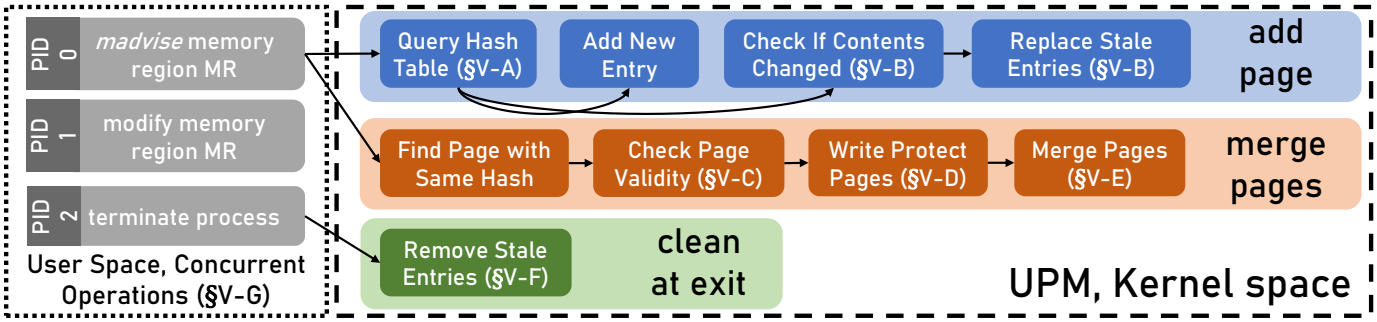


Fig. 4: *UPM* supports concurrent operations on memory pages and exposes methods for managing deduplication.

done entirely on the user side, constituting explicit and opt-in permission for sharing. Page sharing is limited to data explicitly selected by the user, and other tenants cannot detect the contents of memory pages not selected for deduplication.

Since we employ page sharing in *UPM*, it is possible to detect that sharing of a specific page occurs through a side-channel attack [51]. However, a potential leak does not provide any information on functions using the same memory pages, except that such data exists in the system. Moreover, since serverless systems are black-box and do not allow users to control the function placement, attackers can only achieve co-location with another tenant through trial and error.

V. Implementation Details

In this section, we detail the challenges and features of the *UPM* implementation, answering the following questions:

- What hash algorithm and data structures do we use for *madvised* pages? (Sec. V-A)
- How does *UPM* handle *madvise* on the same address, but with different content? (Sec. V-B)
- How does *UPM* handle page modifications before they are deduplicated? (Sec. V-C)
- How does *UPM* prevent page modifications during *UPM* deduplication? (Sec. V-D)
- How does *UPM* implement the page merging? (Sec. V-E)
- How does *UPM* handle cleaning memory pages when processes exit? (Sec. V-F)
- Does *UPM* support concurrent access? (Sec. V-G)

We present an overview of *UPM* operations in Figure 4.

A. Hash Tables

In *UPM*, we search, insert, and delete *madvised* pages, and we do not need to store and retrieve elements in a specific order. Therefore, a hash table is the most suitable structure to store the sharable pages since it enables search and modifications in O(1) time on average. We reuse the well-tested Linux built-in hash table defined in `linux/hashtable.h`. The built-in hash table fulfills all of our requirements, and we can avoid increasing the code complexity in the kernel space. This hash table is implemented as an array of linked lists using separate chaining to resolve hash collisions.

Hash table performance depends on the hash algorithm and its load factor, defined by the ratio of elements to buckets in the table. First, we use the `xxHash` function provided in the Linux kernel because of its high performance. Then, based on profiling results (III), we assume that most serverless functions merge at most 50% of their memory. Furthermore, previous work has shown that 90% of serverless functions consume no more than 400 MB [15]. Therefore, the hash table is set to best perform with *madvised* pages of size up to 200MB. Then, the size is increased by a coefficient of 1.3, a common rule of thumb for hash tables. For page sizes of 4kB, the hash table is set to have $\frac{200MB}{4KB} \cdot 1.3$ different buckets. Since each hash table bucket is just an 8-byte pointer, the hash table adds 520kB of space overhead. The hash table size does not pose a challenge even for larger memory allocations, such as in machine learning models; assuming *madvised* pages of up to 2GB increases the static table size only to 5 MB.

Each bucket entry contains the virtual address (8 bytes), pointers to the page and memory descriptors (8 bytes each), linked list pointers (16 bytes), and the actual hash value to resolve conflicts (8 bytes). Thus, each *madvised* page requires 48 bytes, and the space overhead is 1.17% of deduplicated memory. This overhead per function is at most 3120kB, and it is reached only in the pessimistic case when function containers *madvise* many unshareable pages.

B. Reversed Hash Tables

The same virtual address can be *madvised* multiple times by the user. As long as the contents of the memory page have not changed, *UPM* can ignore subsequent calls. To verify if page contents have changed, we borrow from KSM the concept of *reversed* map, where the usual mapping from hash value to address is reversed. This hash table uses a virtual address (8 bytes) as an index to identify a page entry, and we store the hash value there (8 bytes). Virtual addresses alone cannot uniquely identify a page since different processes may use the same address. Thus, we store the memory descriptor (8 bytes) to identify the process. By querying the reversed map, we detect if the same process has previously *madvised* this memory page with different contents and replace the stale entry. Adding the process identifier (PID, 8 bytes) and linked list pointers (16 bytes), the total size of each entry is 48 bytes, leading to the same space overhead as the previous hash table.

C. Memory Page Validity

Functions can modify memory pages after reporting them as sharing candidates. When *UPM* finds a memory page with contents identical to a new one, we verify if the page is still in the memory by checking the present bit in the page table. Then, we recalculate the hash value to verify that page contents have not changed.

D. Write-protecting Pages

UPM uses copy-on-write to prevent deduplication of pages that have been modified. When *UPM* finds two pages with identical contents, it locks them to prevent the operating system from swapping the pages. To ensure that the contents of the pages are not changed before they are merged, we apply write protection by resetting the corresponding bit in the page table entry. Since the pages may be modified just after we compare hash values and before we write-protect them, we perform a byte-by-byte comparison of contents. Adding write protection to a page does not prevent the page from being modified – rather, write operations on a write-protected page will generate a page fault, and a new physical page will be allocated. Thus, in this unlikely scenario, no page deduplication is possible, but *UPM* will find out that the page has been modified and handle the change correctly.

E. Page Merging

UPM shares memory pages in a copy-on-write fashion. We merge the newly *madvised* page with the page already present in the hash table. First, we replace its *page frame number* (PFN) in the page table entry (PTE) of the new page, making the two virtual pages reference the same physical page. Before changing the PFN, we acquire the page table locks for both pages and flush the cache and the translation lookaside buffer (TLB). This prevents functions from accessing the new page contents by using its old page frame.

After merging pages, we renew the reverse mapping information in *UPM* hash tables. After the pages are merged and marked as write-protected, any write operation will cause a page fault, and a new page frame will be allocated for the modified page. Linux kernel is responsible for processing the page faults, and *UPM* can use this unmodified.

F. Removing Invalid Entries

The status of a memory page can change after it has been *madvised* to the *UPM*. For example, memory pages can be swapped out, freed, or modified, leaving a stale page entry in *UPM* hash tables. However, in the serverless context, the lifetime of such stale entries will be limited by the container's duration. Furthermore, the vast fraction of stable pages constrains the frequency of stale entries. Thus, *UPM* needs only to clear such entries on the function's exit.

We implement a cleaning function to clear all entries associated with a process when it exits. We store a flag for each process to mark if it has added entries to the *UPM*. When a process that has used *UPM* exits, the system iterates over all entries in the reversed hash table to find pages associated with the PID of the process. While it would be simpler to iterate over the entire virtual memory area of the process, this solution would not be sufficient. The addresses of freed memory pages are no longer recorded in the process memory descriptor, and *UPM* could keep stale entries if it had inspected only pages belonging to the process at the time of exit.

G. Concurrency and Consistency

UPM allows for safe modifications of memory pages through the verification mechanisms outlined in previous sections and the write protection. When modifications to the page contents are detected, *UPM* discards this page as a sharing candidate. The byte-by-byte comparison will discover modifications if the page is written to before applying the write-protection. Furthermore, we add a page descriptor comparison right before the page merging to detect page faults. Therefore, the deduplication has no impact on memory safety, and the *madvise* can be conducted concurrently with other operations.

The majority of operations performed by *UPM* are reads needed to calculate hashes and perform comparisons. The read operations of *UPM* have no influence on the page, except for the merging when it changes the references to physical pages. However, flushing caches and TLB will force potential concurrent readers to obtain the physical addresses again from the page table. Thanks to the page table locks acquired during the merge process before flushing caches (Sec. V-E), we ensure that readers will not be able to retrieve page table entries while *UPM* conducts the update. Thus, when other readers finally access the page table, they will obtain the correct reference to a physical page with identical memory content.

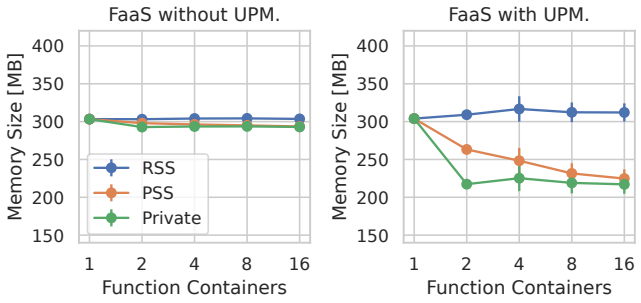
VI. Evaluation

To prove that *UPM* can be used effectively to reduce the memory footprint in FaaS, we evaluate the deduplication with serverless workloads and answer the following questions:

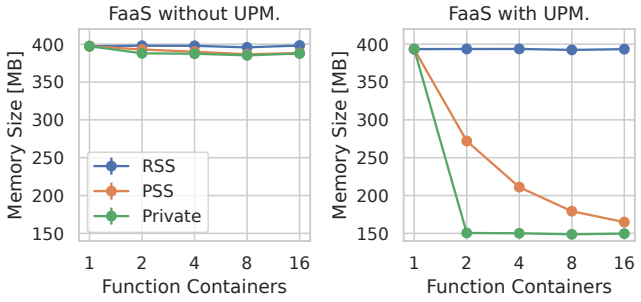
1) Does *UPM* decrease memory footprint of functions (Sec. VI-C)?
2) How much memory can be saved (Sec. VI-D)?
3) How much time overhead does *UPM* add (Sec. VI-E)?
4) How much slower are cold startups in serverless with *UPM* (Sec. VI-F)?
5) Which part of the memory deduplication algorithm takes the most time (Sec. VI-G)?

A. Evaluation Setup

We use two systems to evaluate *UPM*. First, we use a slower System A with a Nehalem CPU in the 45 nm process. Then, we use a higher-performance System B with a Skylake CPU in the 14 nm process. On both systems, the evaluation is performed using a virtual machine running a Linux kernel modified with *UPM*. The x86-64 virtual machine runs Ubuntu 18.04 with a modified Linux kernel 4.15.18, built with GCC 8.3.0. We configure *UPM* to support sharing of up to 2 GB of memory per function. We use Docker 20.10.6 and serverless functions from the benchmark SeBS [12] using Python 3.7.5.



(a) Image recognition with ResNet and PyTorch.



(b) Image recognition with AlexNet and PyTorch.

Fig. 5: Memory deduplication with *UPM*: memory consumption of concurrently executing functions.

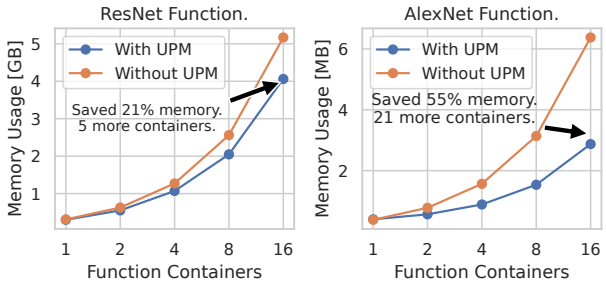


Fig. 6: The effects of *UPM* on memory consumption.

System A The system has four Intel Xeon X7550 CPUs, each one with 8 physical cores at 2.00 GHz frequency, and 1 TB of memory. It runs Debian 10 with kernel 4.19.

System B The system has two Intel Xeon 4110 CPUs, each one with 8 physical cores running at 2.10GHz frequency, and 125 GB of memory. It runs Ubuntu 20.04 with kernel 4.15.18.

B. Benchmarks

We select real-world machine learning inference Python functions to conduct the evaluation from the SeBS benchmark: *image-recognition* that uses the ResNet50 model, and its counterpart *recognition-alexnet*. We use the CFFI library to expose the `madvise` function to Python code. Since the model is not stored directly in a contiguous memory region, we iterate over its components to *advise* all components. We achieve this goal without changing any line of code in PyTorch.

Additionally, we use a microbenchmark to characterize the time overhead of memory deduplication. The function loads different sizes of random data from the **same** file into memory and *madvises* the data. Since all memory pages are distinct, functions can only benefit from inter-process sharing.

C. Memory Usage of Function Containers

To evaluate the effectiveness of our memory deduplication approach, we simulate concurrent executions of serverless functions on a local machine running with the modified Linux kernel (System A). We deploy each function instance in Docker containers that execute concurrently and can benefit from page deduplication. For each container, we run five invocations to stabilize memory consumption and measure the memory consumption of the warm function instance. We measure the resident set size (RSS) defined as the total memory allocation of a process, including zero and shared pages, the private memory size of a process, and the proportional set size (PSS) that adjusts the memory consumption of a process by considering the sharing of memory by n processes:

$$PSS = \frac{shared\,memory}{n} + private\,memory$$

For $n = 1$, PSS is equal to RSS since no memory is shared. When increasing the number of processes n, the influence of the shared part on the memory consumption decreases.

Figure 5 presents the measured memory consumption of serverless benchmark functions for a varying number of concurrently residing warm function instances. Each data point shows the memory size per function container.

UPM reduces the PSS of each container by 14.1% with two concurrent containers and 26.4% with 16 concurrent containers for the ResNet example, and by 29.4% and 55% respectively for the AlexNet example. For AlexNet, *UPM* reduces the private memory size to around 150 MB across the runs, saving about 250 MB of memory on each container.

D. Memory Usage of the System

We measure the memory usage of the entire FaaS system, using the same experimental setting as the memory evaluation on containers in VI-C. We record the memory increase on the system caused by the containers, by executing the Linux command `free -m` before and after launching the containers and invoking the function five times. The system's memory usage can better illustrate the memory reduction effect, since it also counts the data structures that *UPM* creates in the kernel, including the hash tables and the page entries.

Figure 6 shows that for ResNet, *UPM* reduces the total memory usage of the system by 20% with 16 containers running concurrently. The memory reduction reaches 1134MB. Given the PSS of around 225MB for a container, five more containers running the same function can be added to the system. For Alexnet, the reduction is even greater: 55% with 16 containers, amounting to 3585 MB. Given the PSS of about 165 MB per container, this means 21 more containers could be added to the system running the same function.

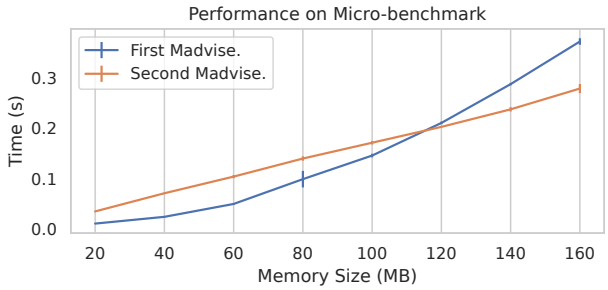
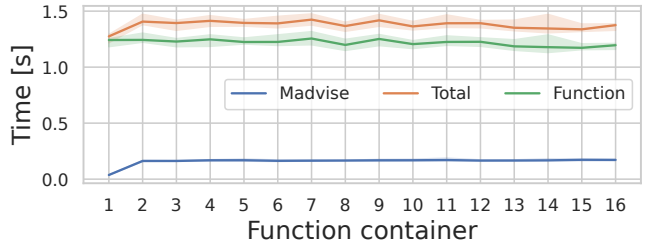
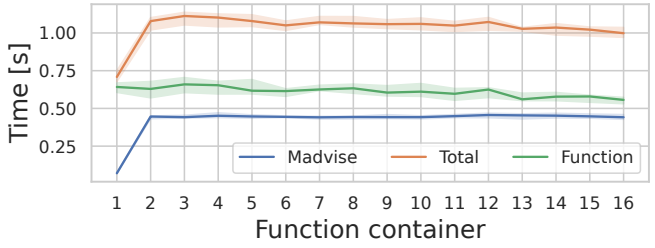


Fig. 7: Adding and sharing memory through *madvise*.



(a) Image recognition with ResNet and PyTorch.



(b) Image recognition with Alexnet and PyTorch.

Fig. 8: Time overhead of memory deduplication on first cold invocation, reported medians with parametric 95% CIs.

E. Time Overhead of UPM

We run the microbenchmark on System B to measure the time impact of sharing memory with *UPM*. Figure 7 shows the results on the micro-benchmark described in Sec. VI-B. Each data point is the average of 10 runs with the standard deviation. The first *madvise* call shows the cost of the first launched program that only adds the pages in the hash table without merging any pages. On the other hand, the second *madvise* call represents the second launched program that also merges the pages.

F. Time Overhead on Cold Startups

To test using *UPM* in a FaaS system, we assume users call the *advice* on function data once, in the first invocation of the function during the cold initialization. In the following experiments, which we repeat ten times, each function is invoked once to measure the time of the `madvise` system call, and the functions are invoked in a cold container. We assume the most pessimistic scenario for the performance of *UPM*, where deduplication is conducted synchronously and the user function blocks until it is finished.

Functionality	Sharing	Sharing & Merging
Search in Hash Table	4.4%	61.4%
Calculate Hash	32.5%	19.6%
Search in Reversed Hash Table	51.2%	10.1%
Merge Pages	0%	6.3%
Spin Locks	5.5%	2.1%
Add Page to Hash Table	9.4%	0.9%

TABLE I: Distribution of *UPM* time across the most important parts of the system, for first function (**Sharing**) and second function (**Sharing & Merging**).

We launch 16 containers running the function and invoke them one by one on System B. The average and the standard deviation of the time cost of the invocation of each container are shown in Figure 8. The figure shows three lines: total time, the time spent on the `madvise` system call, and the time spent for the function call excluding the `madvise` system call.

The overhead introduced by deduplication is equal on average to 12% and 42% of function time on ResNet and Alexnet, respectively. The leap after the first container is explained from page sharing beginning from the second container. However, this overhead applies to the first cold invocation, only once per the entire container lifetime. All future warm invocations do not call the `madvice` function again.

G. Time Overhead Distribution

To analyze the sources of the time overhead *UPM* introduces, we profile the kernel code and measure the execution time of the *image-recognition* function on System A. In this function, we *madvise* approximately 100MB of model memory. Table I presents the breakdown of timing data across most time-consuming functionalities, including the time needed to acquire the spin locks. The main source of CPU overhead in `madvise` is iterating over hash table entries. This is not surprising as *UPM* has to compare each memory page in the shared region with existing hash table entries. One-fifth of the time is spent on hashing memory pages and is limited primarily by the DRAM bandwidth. We observe that the overheads of using spinlocks when modifying hash tables are relatively low during normal execution.

VII. Discussion

In this work, we introduce *UPM*, a memory deduplication system designed for serverless workloads. In the following, we discuss the integration of *UPM* with cloud systems and analyze how memory sharing can be improved through the synergy with other cloud and FaaS technologies.

When to deduplicate? We evaluate the pessimistic scenario of deduplication occurring on the critical path to highlight the overheads of our system (Sec. VI-F). In practical deployments, the deduplication will be moved from the critical path, either to a background thread executing or to the cloud operator side. In the former, users pay the CPU cost of reducing memory consumption in parallel with the function workload. In the

second scenario, users only annotate memory regions at the first invocation, and the cloud operator triggers deduplication logic in the FaaS runtime. This scenario comes with no overhead or costs to the user.

Who benefits from deduplication? In the classic setup of memory deduplication, page sharing is entirely transparent, and users are unaware when their virtual machines share data in the background. All memory savings benefit the cloud provider directly, and users keep paying the same price for requested resources. On the other hand, in the serverless setup, users must take the initiative to enable page sharing, and they can expect performance overheads of deduplication. Therefore, to incentivize users, cloud providers should allow users to participate in the benefits of saving memory resources. Since the billing models of FaaS always include a memory component, cloud providers can adjust the price by subtracting memory savings from the requested memory allocation.

How can the sharing effectiveness of *UPM* be further improved? The application of *UPM* opens new directions for automatically detecting memory likely to be shareable. In many languages, initialization and heap construction can be done at build time [47], allowing static determination of sharing candidates. Furthermore, serverless functions consist of small codebases that are executed thousands of times in the cloud. Such processes can be easily profiled to automatically locate memory pages with identical and constant contents. Similarly to virtual machine fingerprinting and co-location techniques [33, 41], containers with sharing potential can be migrated and co-located on a single machine.

VIII. Conclusions

We propose and implement User-guided Page Merging (*UPM*), a novel memory deduplication approach optimized for FaaS systems. *UPM* improves scanning-based deduplication with user-provided hints to enable deduplication on serverless running for seconds and not hours. Furthermore, *UPM* does not require modifying serverless runtimes, works for all processes handling serverless workloads, regardless of their language and runtime, and extends security guarantees of KSM with controlled sharing. In an evaluation with machine learning inference, we show *UPM* can reduce system memory utilization by more than half, and allows FaaS servers to handle more containers without increasing hardware resources.

Acknowledgment

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 program (grant agreement PSAP, No. 101002047), and EuroHPC-JU funding under grant agreements DEEP-SEA, No. 95560 and RED-SEA, No. 955776). We would like to thank Jike Song for providing valuable advice and support. We would also like to thank the Swiss National Supercomputing Centre (CSCS) for providing us with access to their HPC system Ault.

References

[1] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 193–206.

[2] I. Müller, R. Marroquín, and G. Alonso, "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 115–130.

[3] N. Nikitas, I. Konstantinou, V. Kalogeraki, and N. Koziris, "Cherry: A Distributed Task-Aware Shuffle Service for Serverless Analytics," in *2021 IEEE International Conference on Big Data (Big Data)*, 2021, pp. 120–130.

[4] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "rFaaS: Enabling High Performance Serverless with RDMA and Leases," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 897–907.

[5] M. Copik, R. Böhringer, A. Calotoiu, and T. Hoefler, "FMI: Fast and Cheap Message Passing for Serverless Functions," in *Proceedings of the 37th International Conference on Supercomputing*, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 373–385.

[6] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons: Lambdas on a Diet," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 45–59.

[7] D. Saxena, T. Ji, A. Singhvi, J. Khalid, and A. Akella, "Memory Deduplication for Serverless Computing with Medes," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 714–729.

[8] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 559–572.

[9] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the Curtains of Serverless Platforms," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 133–145.

[10] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana, "OFC: An Opportunistic Caching System for FaaS Platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 228–244.

[11] P. García-López, M. Sánchez-Artigas, S. Shillaker, P. Pietzuch, D. Breitgand, G. Vernik, P. Sutra, T. Tarrant, A. Juan-Ferrer, and G. París, *Trade-Offs and Challenges of Serverless Data Analytics*. Cham: Springer International Publishing, 2022, pp. 41–61.

[12] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. Association for Computing Machinery, 2021.

[13] S. Shillaker and P. Pietzuch, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 419–433.

[14] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful Functions-as-a-Service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, jul 2020.

[15] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 205–218.

[16] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[17] J. Jarachanthan, L. Chen, F. Xu, and B. Li, *AMPS-Inf: Automatic Model Partitioning for Serverless Inference with Cost Efficiency*. New York, NY, USA: Association for Computing Machinery, 2021.

[18] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1049–1062.

[19] U. Elordi, L. Unzueta, J. Goenetxea, S. Sanchez-Carballido, I. Arganda-Carreras, and O. Otaegui, "Benchmarking Deep Neural Network Inference Performance on Serverless Environments With MLPerf," *IEEE Software*, vol. 38, no. 1, pp. 81–87, 2021.

[20] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, "Machine Learning at Facebook: Understanding Inference at the Edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 331–344.

[21] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing," *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 447–457, 2020.

[22] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance Optimization for Edge-Cloud Serverless Platforms via Dynamic Task Placement," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 41–50.

[23] B. Wang, A. Ali-Eldin, and P. Shenoy, "LaSS: Running Latency Sensitive Serverless Computations at the Edge," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 239–251.

[24] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang, "When Serverless Computing Meets Edge Computing: Architecture, Challenges, and Open Issues," *IEEE Wireless Communications*, vol. 28, no. 5, pp. 126–133, 2021.

[25] G. Miłoś, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened Page Sharing," in *2009 USENIX Annual Technical Conference (USENIX ATC 09)*. San Diego, CA: USENIX Association, Jun. 2009.

[26] 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 *USENIX Annual Technical Conference*, 2018.

[27] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers," ser. USENIX ATC '18. USENIX Association, 2018.

[28] K.-T. A. Wang, R. Ho, and P. Wu, "Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019.

[29] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, p. 412–447, nov 1997.

[30] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proceedings of the Linux Symposium*, 2009, pp. 19–28.

[31] C.-R. Chang, J.-J. Wu, and P. Liu, "An Empirical Study on Memory Sharing of Virtual Machines for Server Consolidation," in *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, 2011, pp. 244–249.

[32] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, 2003.

[33] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, p. 27–36, jul 2009.

[34] P. Sharma and P. Kulkarni, "Singleton: System-Wide Page Deduplication in Virtual Environments," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing - HPDC '12*. ACM Press, 2012.

[35] S. Rachamalla, D. Mishra, and P. Kulkarni, "Share-o-meter: An empirical analysis of KSM based memory sharing in virtualized systems," in *20th Annual International Conference on High Performance Computing*, 2013, pp. 59–68.

[36] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "XLH: More Effective Memory Deduplication Scanners Through Cross-layer Hints," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 279–290.

[37] J. F. Kloster, J. Kristensen, and A. Mejlholm, "Determining the use of Interdomain Shareable Pages using Kernel Introspection," *Department of Computer Science, Aalborg University*, 2007.

[38] L. You, Y. Li, F. Guo, Y. Xu, J. Chen, and L. Yuan, "Leveraging Array Mapped Tries in KSM for Lightweight Memory Deduplication," in *2019 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2019.

[39] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao, "CMD: Classification-Based Memory Deduplication through Page Access Characteristics," *SIGPLAN Not.*, vol. 49, no. 7, p. 65–76, mar 2014.

[40] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference Engine: Harnessing Memory Redundancy in Virtual Machines," *Commun. ACM*, vol. 53, no. 10, p. 85–93, Oct. 2010.

[41] M. Sindelar, R. K. Sitaraman, and P. Shenoy, "Sharing-Aware Algorithms for Virtual Machine Colocation," in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 367–378.

[42] "Effects of Memory Randomization, Sanitization and Page Cache on Memory Deduplication."

[43] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman, "An Empirical Study of Memory Sharing in Virtual Machines," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 273–284.

[44] F. Vano-Garcia and H. Marco-Gisbert, "KASLR-MT: Kernel Address Space Layout Randomization for Multi-Tenant cloud systems," *Journal of Parallel and Distributed Computing*, vol. 137, pp. 77–90, 2020.

[45] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 467–481.

[46] 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.

[47] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, "Initialize Once, Start Fast: Application Initialization at Build Time," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019.

[48] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient Serverless Inference through Tensor Sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022.

[49] B. Lee, S. M. Kim, E. Park, and D. Han, "MemScope: analyzing memory duplication on android systems," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, 2015, pp. 1–7.

[50] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, p. 79–95, dec 1995.

[51] 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.