SnapStore: A Snapshot Storage System for Serverless Systems

Abhisek Panda
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India
abhisek.panda@cse.iitd.ac.in

Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India
srsarangi@cse.iitd.ac.in

ABSTRACT

Serverless computing is getting increasingly popular because of its fine-grained billing model and autoscaling features. To speed up the process of functions' sandbox creation, cloud providers typically utilize snapshot and restore-based mechanisms for pre-warmed snapshots. This effectively trades off the startup latency with the storage requirements and the overhead of creating/restoring these snapshots. Hence, there is a need to compress the snapshots by identifying identical data chunks across snapshots and then design methods to quickly deduplicate snapshots and retrieve them. We propose SnapStore – a novel method of finding such *duplicates*. As opposed to conventional work that relies on better hashing methods, we use the natural structure of the program's memory map to reduce wasted work during deduplication. Furthermore, we sequentialize and minimize disk accesses as much as possible while retrieving a snapshot into a RAM-based cache. Both of these optimizations, yield a reasonably large speedup in the deduplication process as compared to the state-of-the-art (≈ 46% in the snapshot deduplication time and ≈ 82.6% in the retrieval time on HDDs). Upon integration with FaaSnap (a state-of-the-art serverless platform), SnapStore improves the end-to-end latency of serverless functions by 25.9% along with 2.4× storage space reduction over vanilla FaaSnap on HDDs. With SSDs, our deduplication time and retrieval time reduce by 36.2% and 75.8%, respectively, with almost no degradation in the end-to-end latency.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Information systems → Deduplication**; • **General and reference** → *Performance*; • **Software and its engineering** → *Virtual memory*.

KEYWORDS

Serverless computing, Function-as-a-Service, Deduplication, Snapshot storage systems

ACM Reference Format:
Abhisek Panda and Smruti R. Sarangi. 2023. SnapStore: A Snapshot Storage System for Serverless Systems. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3590140.3629120

1 INTRODUCTION

Serverless computing is emerging as a popular cloud computing paradigm [20, 22]. Over the last few years, several prominent cloud providers including Amazon [29], Google [9], IBM [10], and Microsoft [4] have adopted this paradigm due to its superior billing model and auto-scaling features. A serverless platform primarily comprises two services: Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) [13]. In FaaS, a user provides a platform with a series of functions that represent different parts of a larger task. On the other hand, BaaS replaces server-side components such as file storage, authentication, and databases with off-the-shelf services. The platform deploys a set of sandboxes across distributed nodes to execute each of these functions based on the functions' arrival rates thereby providing autoscaling features. Moreover, the platform imposes a pay-as-you-go billing model by charging a user for the duration its functions are being executed. Due to these features, a large number of applications from the domains of machine learning, data analytics, and IoT are adopting this paradigm [11, 25].

In FaaS, platforms create a sandbox environment in order to execute a function. Therefore, the cost of sandbox creation is on the critical path [3, 12, 20, 42, 43, 47]. Prior studies have shown that approximately 50% of serverless functions execute in less than one second [41]. Therefore, the cost of creating a sandbox has a significant impact on the latency of a function (referred to as *cold start*). Prior work proposed the following techniques to mitigate the deleterious effects of cold starts: reusing a sandbox [26, 46], sharing a sandbox [2], utilizing lightweight sandbox mechanisms [1] and restoring from a sandbox's stored snapshot [3, 12, 42, 47].

It is widely believed that restoring a sandbox from its snapshot is an effective cold start mitigation technique [3, 42]. In this technique, platforms serve future requests of a function by restoring a sandbox from its snapshot. As a result, they eliminate the cost of library loading and runtime initialization [12, 47]. If a platform deploys hundreds of serverless functions on a host and uses this method, then it can lead to large storage requirements [3, 42]. Therefore, prior work proposed storing function snapshots on a remote storage service [47]. However, the cost of fetching a function snapshot from a remote storage service depends on the function snapshot's size and network bandwidth, thereby limiting the benefits of snapshot and restore-based mechanisms.

In the context of cloud data, prior work [8, 15, 23, 35] has utilized deduplication mechanisms to minimize storage requirements. Typically, a deduplication mechanism consists of two stages: duplicate elimination and retrieval [30, 51]. In the duplicate elimination stage, we eliminate redundant data chunks from files. In the retrieval stage, we reassemble a file from its constituent data chunks. Our proposed scheme *SnapStore* utilizes a deduplication mechanism to store a large number of function snapshots on a machine, thereby

Middleware '23, December 11–15, 2023, Bologna, Italy

minimizing the number of remote storage accesses. Given that a serverless platform executes hundreds of functions on a machine, adding a deduplication mechanism might lead to high CPU utilization [1, 38]. Therefore, our primary constraint is that the duplicate elimination stage must be lightweight and also off the critical path. Before restoring the sandbox of a function, platforms must quickly reassemble its snapshot, which is on the critical path.

This is an established problem and there are many solutions for fast deduplication and retrieval of snapshots. The solutions in prior work [21, 30, 40, 51] revolve around breaking a snapshot file into fixed or variable sized chunks, identifying duplicates based on hashing and storing as few unique chunks as possible. There is a high degree of redundancy across snapshots because they share similar kernel modules, software packages, libraries and sometimes initialization files. Also, the account statistics of Amazon Lambda show that 53% of users utilize Node.js and 36% utilize Python [39]. Hence, the problem essentially boils down to efficiently managing a dictionary of chunks and appropriately storing the chunks on disk. Prior solutions propose fairly generic mechanisms and the area of efficiently managing such dictionaries during the deduplication/retrieval process is fairly mature.

We adopt a paradigmatically different approach in this paper. Our key observation is that the deduplication process is not equally productive for all the memory regions of a process (whose snapshot is captured). There are regions such as the read-only pages, memory-mapped pages (with read permission), and pages that are not a part of the main runtime process, which have a high degree of overlap with other snapshots. Whereas, pages that are written to by the runtime process have very little degree of overlap. There is thus no need to waste time and space in checking for duplicates in these regions. If we focus on the regions that are expected to be more productive we can use smaller hash tables and also organize the chunks in a more effective manner for the entire snapshot. Before a function's execution, we reassemble its snapshot on a RAM-based cache (implemented using RAMFS) by retrieving unique chunks from the disk or previously reassembled snapshots. For each chunk, we fill all the snapshot's memory locations that store the chunk. This approach ensures that our disk accesses are minimal and nearly sequential, which is very beneficial. Note that SnapStore divides the operations efficiently so that multiple threads can handle each task concurrently and atomically (crash consistent).

To summarize, our contributions are as follows:

(1) We perform a comprehensive study of a function's snapshot to identify suitable memory regions for deduplication.
(2) We design SnapStore, which performs selective deduplication of a snapshot – different memory regions are treated *differently*. This reduces useless work and gives us performance benefits.
(3) Specifically, we improve the deduplication time by at least 46% compared to other state-of-the-art approaches on HDDs while limiting the degradation in the deduplication ratio by 10%.
(4) We improve the retrieval time by at least 82.6% compared to other state-of-the-art approaches on HDDs by ensuring *minimal and sequential* disk accesses.
(5) We improve the end-to-end latency of serverless functions by 25.9% over FaaSnap (a state-of-the-art serverless solution) along with achieving a 2.4× storage space reduction on HDDs.
(6) We get a benefit on SSDs because of better prefetching (36.2% and 75.8% reduction in dedup and retrieval times, respectively).

The rest of the paper is organized as follows. We discuss the relevant background for the paper in Section 2. Subsequently, we perform a comprehensive analysis of a function snapshot in Section 3. This is followed by the design of SnapStore in Section 4. Section 5 evaluates the effectiveness and robustness of SnapStore. We discuss the related work in Section 6, and finally, we conclude in Section 7.

2 BACKGROUND

2.1 Serverless Computing

The serverless computing paradigm reduces operational complexity and provides a fine-grained billing scheme. In FaaS, a user decomposes an application into pieces of code snippets (*functions*) and provides them to a serverless platform along with the order of execution. Upon receiving a request from an external entity (a user or an event), the platform spawns an ephemeral sandbox containing the corresponding function and starts to execute the function within the sandbox. Typically, spawning a sandbox involves booting the sandbox and loading its memory state, code and libraries. Due to security concerns, sandboxes cannot communicate among themselves; thus, they utilize a remote storage engine for data communication. Now, at execution completion, the provider typically destroys the sandbox in unoptimized implementations. For each request, the spawning of an ephemeral sandbox is on the critical path (referred to as *cold start*) – this is a performance overhead.

To mitigate the cold start problem, existing frameworks employ the following techniques: light-weight sandbox mechanisms [1], warm containers [26, 46] and better snapshot-restore techniques [3, 12, 47]. Warm containers keep sandboxes in the system for a specified period of time rather than destroying them at execution completion. However, this causes the system to waste resources. In contrast, snapshot-restore mechanisms take a snapshot of a sandbox after the OS, runtime and libraries have been initialized. They then restore a function sandbox from the snapshot, thus omitting the startup step [3, 12]. Commercial serverless platforms such as Amazon Lambda use this mechanism to mitigate the cold start problem [5]. Amazon isolates function execution on a host machine using Firecracker microVMs.

In this paper, we use Amazon Lambda's Firecracker to manage our microVMs [1, 3, 47]. Let us discuss its design.

2.2 Firecracker

Firecracker is a virtual machine manager (VMM) that uses KVM to supervise the execution of a guest operating system on the host machine [1]. It spawns and manages the virtual CPUs assigned to the guest as threads. It allocates a memory region for the guest by invoking the *mmap* system call. The data communication between the guest and the host takes place through a ring buffer (*virtqueue*) [17]. To take a snapshot of the guest in the *pause* state, Firecracker creates the following files: the microVM state file and the memory file. The size of the microVM state file is about 15 KB because it stores the VM metadata, the vCPU state, the memory state, the device state and the KVM state. The memory file, on the other hand, has the same size as the VM because it stores the contents of the memory area of the microVM. As a result, the memory file contributes to the storage overhead of a snapshot.

2.2.1 Memory Manager. By using the *mmap* system call, the firecracker VMM creates a memory region for a microVM with the specified length, memory protection and flags. The memory region is then registered with KVM along with the *KVM_MEM_LOG_DIRTY_PAGES* flag using the *ioctl* system call. The aforementioned flag instructs KVM to keep track of writes to the memory region [24]. To record writes to the memory region, Firecracker creates an atomic bitmap at the page level (referred to as *dirty_bitmap*). On the completion of an event in the *virtqueue*, the bitmap indexes for the appropriate memory pages are set (referred to as *dirty* memory pages in this paper). Memory pages with an unset bitmap index, on the other hand, are considered *non-dirty* memory pages. Non-dirty memory pages are often zero pages (pages filled with zeros). If the memory snapshot file contains a large number of non-dirty memory pages, storage space is effectively wasted. This gives us an idea: **a bespoke data structure to efficiently store the contents of a microVM's memory space**.

2.3 Deduplication

To improve the storage space utilization of a system, current storage systems employ chunk-based deduplication mechanisms (see Figure 1). These storage systems have two stages: duplicate elimination and retrieval. In the duplicate elimination stage, we eliminate redundant data chunks from a set of files and only store unique data chunks in fixed-size blocks on the disk (referred to as *dblocks*). During the retrieval stage, we restore a deduplicated file from its constituent data chunks present on the disk.

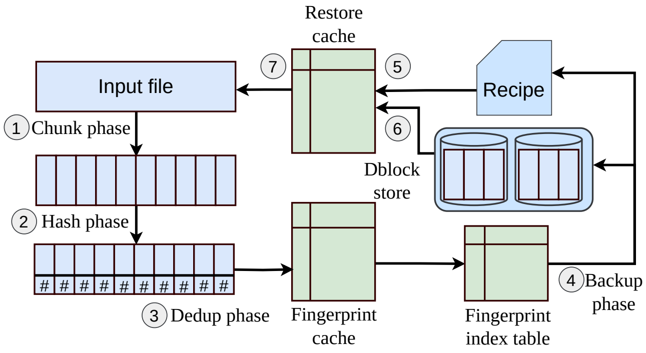


Figure 1: The high-level design of a traditional deduplication system.

Specifically, the duplicate elimination stage consists of four phases: ① the chunk phase, ② the hash phase, ③ the dedup phase, and ④ the backup phase (see Figure 1). During the *chunk phase*, chunk creation algorithms such as Rabin-based CDC [34], TTTD chunking [14], fixed-size or FastCDC [49] divide a file into chunks that either have a fixed or variable size. The subsequent *hash phase* employs a strong hash function such as SHA-1 to compute the hash of each chunk (referred to as the fingerprint) [30, 51]. The *dedup phase* then identifies duplicate data within a file by comparing the fingerprints to entries already stored in a fingerprint index table. A fingerprint index table is a hash table with the chunk's fingerprint as the key and the 4-byte dblock id (d_id) as the value.

Table 1: Workloads used in this paper (adapted from FunctionBench [25]). *Note: The memory column denotes the size of the microVM allocated to a function.*

Workloads	Description	Memory
chameleon	Renders an HTML table.	128 MB
helloworld	Prints "Hello world"	128 MB
image_rotate	Rotates an image by 90 degrees.	128 MB
json_serdes	Performs serialization and deserialization of a JSON file.	128MB
pyaes	Performs AES block-cipher encryption.	128 MB
face_detection	Annotate faces in a given video using the Haar feature-based cascade classifier. [37]	256 MB
matmul_fb	Performs matrix multiplication.	256 MB
video_processing	Performs grayscale conversion of a given video.	256 MB
lr_serving	Predicts the target value for a given data record.	512 MB
lr_training	Trains a linear regression model based on a provided dataset.	512 MB

However, as the data size increases, so does the size of the fingerprint index table. Thus, a need arises to store a part of it on disk and the actively used part in an in-memory fingerprint cache. Finally, the *backup phase* concludes the duplicate elimination stage by storing the unique data chunks on the disk in fixed-size blocks, typically 4 MB. For data recovery, the backup phase generates a *recipe* of a file that contains a list of entries corresponding to data chunks of the file.

To retrieve the file from a recipe, we need to iterate through the recipe's entries (see ⑤ in Figure 1). We then use the dblock's id in the recipe to locate the dblock that contains the chunk's data. Then, we restore the desired data by accessing the dblock (see ⑥ in Figure 1). This process may entail frequent disk accesses [23, 35]. To mitigate this overhead, existing mechanisms employ the following techniques: caching or rewriting. In the caching technique [30], we maintain a restore cache, which contains the recently accessed chunk's fingerprint and chunk's data. On the other hand, the rewriting technique creates redundant copies of chunks in different dblocks for faster access [31, 36]. Finally, the system retrieves the file by piecing together its chunks (see ⑦ in Figure 1).

3 CHARACTERIZATION OF A FUNCTION SNAPSHOT

3.1 Evaluation Methodology

We evaluate a serverless platform on a server-based system and study the characteristics of functions' snapshots. The platform can support functions whose memory footprint ranges between 128 MB and 10,240 MB [28]. The workloads comprise popular real-world serverless functions taken from the FunctionBench [25] suite, which are summarized in Table 1. The system configuration is shown in

Table 2: System configuration

Hardware settings			
Processor	Intel Xeon 6226R CPU, 2.90 GHz		
CPUs	1 Socket, 16 cores	DRAM	256 GB
HDD	Seagate 2TB SATA-3 HDD, 7200RPM		
SSD	Samsung 860 EVO 500GB SATA-3 SSD		
NVMe	Dell M.2 2230, PCIe Gen 3 NVMe (256 GB)		
NAS Server	HPE StoreEasy 1660, ping-time-to-server=0.4 msec		
Software settings			
Guest Linux kernel	5.15	GCC version	7.5
Python	3.10	Firecracker version	1.0.0

Table 2. We store a function snapshot in local storage (similar to Reap [47]). All our experiments deploy one Firecracker [1] microVM instance with two vCPUs and corresponding memory to run a function (see Table 1). *Note: The memory size of the microVM allocated to a function is defined as per its input size.*

Typically, a function snapshot consists of two files: the microVM state file and the memory file (discussed in Section 2.2). Since the snapshot is taken prior to the execution of a function, we analyze the characteristics of non-dirty memory pages stored in the memory file (discussed in Section 3.2) – zeroed pages that are not touched by the OS or other runtime initialization routines.

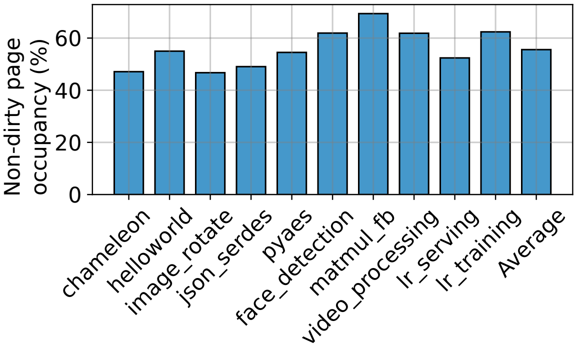


Figure 2: The occupancy of non-dirty memory pages in the memory file of a function snapshot. *Note: A snapshot contains significant amount of non-dirty memory pages that leads to storage wastage.*

3.2 Analyzing Non-dirty Memory Pages

To analyze the occupancy of non-dirty memory pages, we traverse the *dirty_bitmap* to find the memory areas of the microVM that contain dirty memory pages (discussed in Section 2.2). Subsequently, we record memory areas as an array of tuples with the values $\langle offset, len\rangle$, where *offset* denotes the starting location of a memory area from the beginning of the memory file and *len* represents the size of the area – this is on the lines of classical run-length encoding. In Figure 2, we show that 55.6% of the memory file contains non-dirty memory pages, on an average. For a memory-intensive application, the percentage of non-dirty memory pages can be up to 69.4% of the memory file (see Figure 2). This is because the snapshot is taken prior to the function's execution, so the data that will be touched by the main runtime process (one that will execute the function) is not a part of the memory file yet. This motivates us to treat and store the memory snapshot file as a *sparse file* [18].

(1) The occupancy of non-dirty memory pages of a function snapshot is 55.6% of the memory file.
(2) For a memory-intensive application, the percentage of non-dirty memory pages can be up to 69.4% of the memory file.

3.3 Analyzing Dirty Memory Pages

In a serverless computing paradigm, a microVM provides an isolated environment to execute a function. To analyze the amount of memory occupied by a function, we split the dirty memory pages into two regions: runtime (RT) and non-runtime (NRT). The runtime memory region includes the memory pages that belong to the function, whereas the non-runtime memory region includes the memory pages of other processes except the function. Before performing the analysis, we drop the non-dirty memory pages of the function's memory file (referred to as the *compressed function snapshot* in this paper).

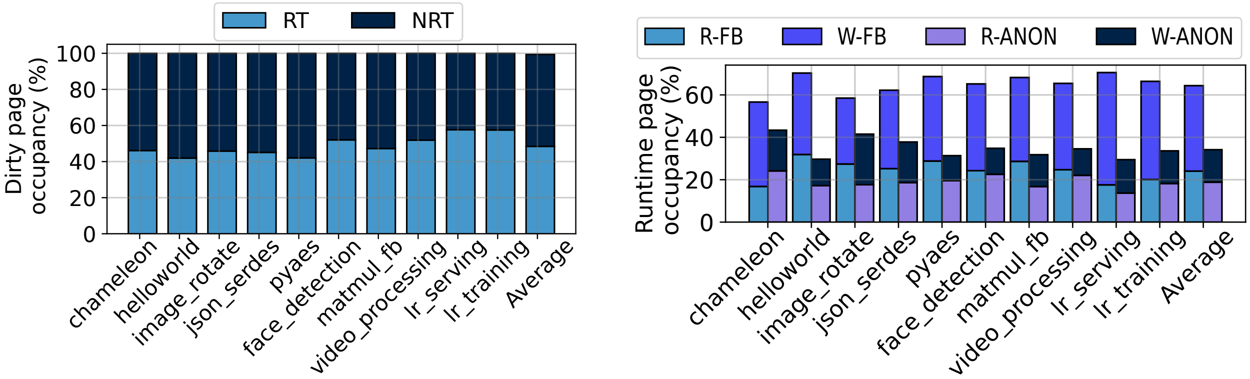
To find the runtime memory regions, we read the *maps* and *pagemap* files of the runtime process present in the *proc* filesystem. In Figure 3a, we show that RT occupies 48.4% of the compressed memory snapshot file, whereas NRT occupies the remaining space, respectively. In the case of lr_serving, face_detection, video_processing, and lr_training functions, the size of RT is larger than NRT because they utilize vision and machine learning Python packages. Let us now analyze the runtime memory pages.

We can divide runtime memory regions into two categories: file-backed (FB) and anonymous ($ANON$). FB refers to memory mapped pages that are part of a file, while the remaining pages (stack/heap/data/bss) are $ANON$. We further divide each category X based on permission: $R-X$ and $W-X$. For instance, $W-X$ represents the memory-mapped pages of category X for which the runtime process has write permission. In Figure 3b, we show that 65.5% of runtime memory pages contain FB pages and 34.5% contain $ANON$ pages. Memory mapped pages with write permission ($W-ANON$ and $W-FB$) are present in 55.6% of the runtime memory regions.

(1) The occupancy of non-runtime memory regions in a function snapshot is 51.6% of the compressed memory snapshot file.
(2) The memory-mapped pages for which a runtime process has write permission occupy about 26.9% of the compressed memory snapshot file.

3.4 Analyzing Duplicate Memory Pages

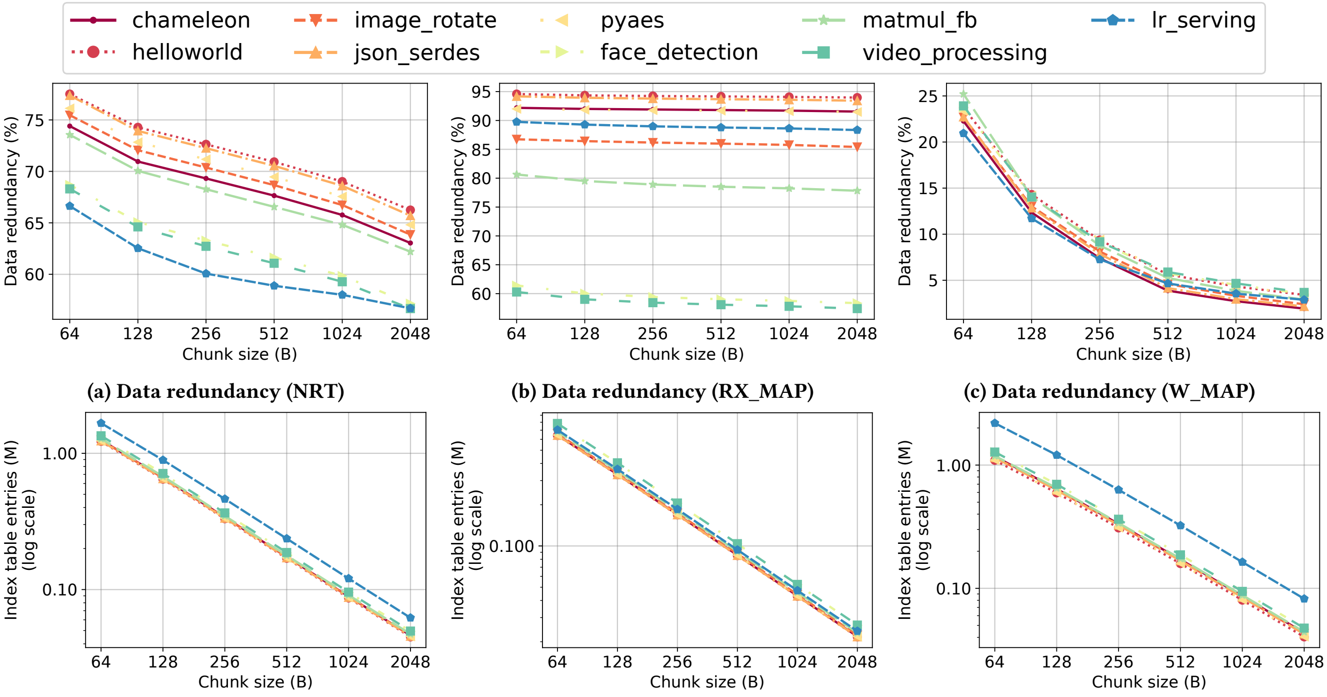
In this section, we discuss the scope of data deduplication mechanisms by finding the amount of duplicate data between two compressed function snapshots: the data redundancy of a function snapshot file $\mathcal{B}$ with respect to a function snapshot file $\mathcal{A}$ is the percentage of duplicate data present in $\mathcal{B}$. For fine grained analysis, we find data redundancy for the runtime and non-runtime memory regions separately.



(a) The occupancy of the runtime and non-runtime memory pages in compressed function snapshots.

(b) The occupancy of file-backed and anonymous memory mapped pages in the runtime memory pages of a function.

Figure 3: Breakup of the dirty-memory pages in the memory snapshot file of a function.



(a) Data redundancy (NRT)

(b) Data redundancy (RX_MAP)

(c) Data redundancy (W_MAP)

(d) Index table size (NRT)

(e) Index table size (RX_MAP)

(f) Index table size (W_MAP)

Figure 4: The data redundancy and total number of entries in the fingerprint table of the following regions of a serverless function: NRT, RX_MAP, and W_MAP with respect to the lr_training function by using the fixed-sized chunk algorithm and varying the chunk size from 64 bytes to 2048 bytes. ***Note: The NRT and RX_MAP memory regions show high memory redundancy, while W_MAP shows low memory redundancy.***

In the case of runtime memory regions, we further divide the memory pages based on the access permissions into two categories: RX_MAP and W_MAP. Memory-mapped pages with write permission are highly unlikely to be duplicated across different serverless functions. W_MAP corresponds to these memory-mapped pages, while RX_MAP contains the remaining pages. RX_MAP mainly contains the shared anonymous pages and file pages. Let us estimate the data redundancy between two compressed function snapshots with respect to the following memory regions: NRT, RX_MAP and W_MAP.

After extracting the memory pages corresponding to a memory region, we split the pages into fixed chunks, whose sizes range from 64 bytes to 2048 bytes [19, 21]. For ease of analysis, we then estimate the data redundancy of a snapshot with respect to the lr_training function (representative) snapshot using the deduplication mechanism (described in Section 2.3). In Figure 4, we show that

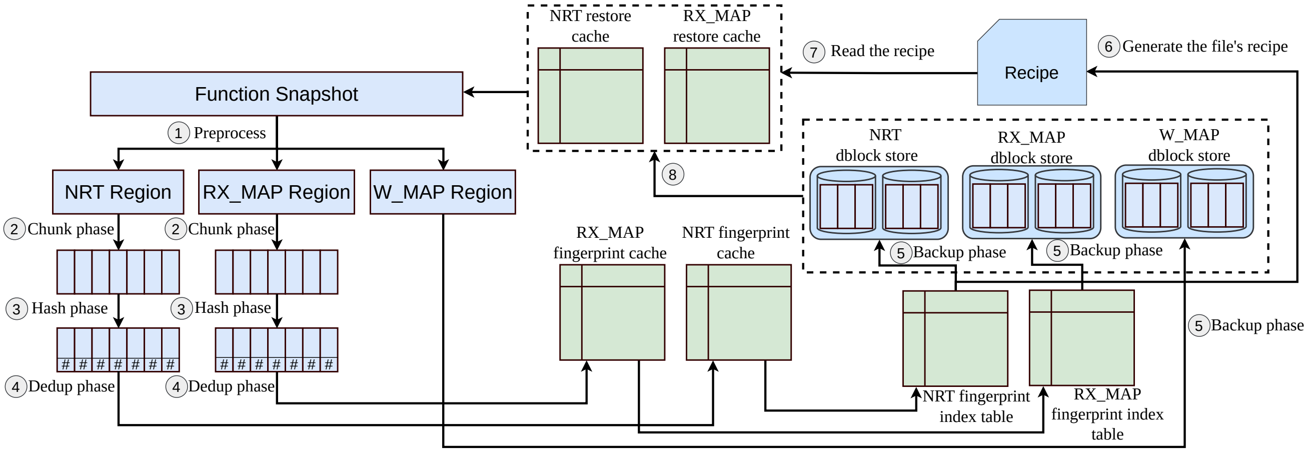


Figure 5: The high-level design of SnapStore.

the data redundancy between the NRT memory regions can range between 56.7% and 77.5% depending on the chunk size. Similarly, the memory redundancy between the RX_MAP memory regions ranges from 94.6% to 57.4%. However, the memory redundancy between the W_MAP regions is up to 25.2%, which is lower when compared to the other memory regions. Therefore, we can conclude that the non-runtime (NRT) and RX_MAP memory regions show high memory redundancy because they store the shared anonymous pages, file pages, and data of other processes that are common across functions with the same runtime environment and host operating system. In Figure 4b, the overlap of the video_processing and face_detection workloads is about 60% because they utilize the *OpenCV* [7] Python library, which is not a part of the lr_training workload.

To estimate the storage overhead of the fingerprint table during deduplication, we measure the total number of entries in the table. In Figure 4, we show that by decreasing the chunk size, the data redundancy increases but the total number of entries in the fingerprint table also increases by up to 27×. Therefore, **the choice of the chunk size in a deduplication scheme is non-trivial because of its memory region dependence and the fact that the fingerprint table can become quite large**.

(1) In the case of runtime memory pages with write permission, there is minimal data redundancy. The rest of the regions have varying levels of redundancy.
(2) To achieve high data redundancy with low storage overhead, the selection of the chunk size is memory region dependent and is quite challenging.

4 DESIGN

In this section, we shall discuss the design of SnapStore, a snapshot storage system that employs a deduplication mechanism to facilitate the storage of a large number of function snapshots. A FaaS platform stores function snapshots on disk. SnapStore reads these snapshots and removes any redundant data. Subsequently, it stores the unique data on the disk in fixed-size blocks (referred to as *dblocks*) that are common to all function snapshots and generates a recipe for each snapshot. When a serverless platform requests a function's snapshot, it reassembles the function snapshot on a RAM-based cache (implemented using RAMFS) by reading the recipe and dblocks either from the disk or the RAM-based cache (previously reassembled snapshots). A serverless platform then uses the reassembled snapshot to spawn a sandbox.

4.1 Design Principles

Serverless computing platforms deploy hundreds of sandboxes on a single server to serve requests for different serverless functions [1]. However, the CPU and memory resources of a server are fixed and often prove to be on the lower side – this results in high resource contention [38]. Furthermore, platforms employ snapshot and restore-based mechanisms to mitigate the cold start latency, where snapshot creation is not on the critical path but the restore is. By introducing a deduplication mechanism along with snapshot creation, the degree of resource contention on platforms sadly increases even further because of higher CPU involvement. Therefore, the proposed deduplication mechanism must have a lower duplicate elimination time and snapshot retrieval time. To improve the aforementioned aspects, we propose a deduplication scheme that exploits the memory region information (mapping between a memory page and its memory region) of a function snapshot. Our idea works because we demonstrated that the degree of data redundancy is memory region-dependent (as discussed in Section 3.4).

4.2 High-level System Design

In Figure 5, we show the design overview of SnapStore. Before creating a snapshot of a microVM, we execute a code snippet that captures information about the microVM's dirty memory regions. On snapshot creation, we remove non-dirty memory pages from a function's snapshot. This is because storing non-dirty memory pages leads to storage space wastage. Subsequently, we divide a function snapshot into three memory regions: NRT (non-runtime memory regions), RX_MAP (function's memory mapped pages without write permission) and W_MAP (function's memory mapped pages with write permission) by using the memory region information of a function snapshot (see Section 4.4 and also see ① in Figure 5). We then consider the NRT and RX_MAP memory regions as candidates for deduplication. Subsequently, we employ the fixed-size chunking algorithm to divide the memory regions into fixed-size chunks with a suitable chunk size depending on its memory region (②). To identify duplicate data exclusive to a memory region category, we compute the hash of the chunk's data and compare it with previous hashes stored in the in-memory fingerprint cache and on-disk index table (stores the full fingerprint table) corresponding to each memory region (③ and ④). Consequently, the cost of duplicate identification decreases significantly.

SnapStore stores the unique data chunks corresponding to a memory region of a function snapshot in separate dblocks (see ⑤ in Figure 5). In comparison to traditional deduplication mechanisms, we have three types of dblocks: NRT, RX_MAP and W_MAP. NRT and RX_MAP dblocks store the unique data chunks of their respective memory regions. In contrast, W_MAP dblocks store the original data chunks that correspond to the W_MAP memory region (discussed in Section 3.4). As a consequence, the data chunks of the memory regions exhibit a degree of spatial locality (within a memory region). For data retrieval, the system generates a recipe of the file (see ⑥).

During the retrieval stage, SnapStore reconstructs a function snapshot on a RAM-based cache by reading the unique data chunks from the disk, or previously restored function snapshots present in the cache. During reconstruction, SnapStore fills the unique data chunks memory region-wise instead of going by the order of entries in the recipe (conventional method) (see ⑦ and ⑧ in Figure 5). This is because dblocks store the unique data chunks of a memory region and this method has an inherent amount of spatial locality. As a result, SnapStore ensures minimal and near-sequential disk accesses, thereby enhancing the performance of the retrieval stage, and enabling more efficient prefetching.

4.3 Data Structures

Table 3 lists the data structures utilized by SnapStore. After dividing a function snapshot into chunks, SnapStore stores every chunk of a function snapshot using the *struct* data type. The members of the chunk's *struct* are the size of the chunk, duplicate flag, dblock ID, chunk type, position (location of the chunk in a function snapshot), 20-byte fingerprint and chunk data. The chunk_type attribute only stores three values: 0, 1, and 2, which represent the NRT, RX_MAP and W_MAP memory regions, respectively. The size of the position member is 8 bytes because the maximum possible size of a function's snapshot is 10GB [28]. To identify duplicates, SnapStore uses per-memory region fingerprint caches and index tables that are common across all the function snapshots.

After removing duplicate chunks, SnapStore stores unique data chunks in a per-memory region dblock_store, which is a linked list of dblocks. Each dblock is typically 4 MB in size and contains the *d_type* attribute to indicate the type of constituent chunks. We split a dblock into two regions: meta and data. The meta region stores the metadata of the constituent chunks, while the data region stores the chunk's data. The dblock_data member of a dblock's metadata stores the dblock_offset value for each chunk, which represents the offset from the beginning of the data region at which the chunk's data is stored. The following attributes determine the size of the meta region: the size of the dblock_data map to store *n* chunks, the d_id and the d_type, where the size of the data region is *n* × chunk size. Note that the value of *n* should be chosen such that the combined size of the meta region and data region is as close as possible to 4 MB (yet less than it, this is the dblock size). A dblock is said to be *full* when it cannot accommodate any more chunks.

Table 3: List of data structures used by SnapStore

Entity	Data type	Description
chunk	struct	Members: 4-byte c_size, 1-byte duplicate_flag, 4-byte d_id, 1-byte c_type, 8-byte position; char fp[20]; unsigned char *data;
fingerprint cache	Linked list	Element: ⟨ 20-byte chunk's fingerprint, 4-byte dblock_id ⟩
index table	Hash table	Key: 20-byte chunk's fingerprint, Value: 4-byte dblock_id
dblock_data	Hash table	Key: 20-byte chunk's fingerprint, Value: 4-byte dblock_offset
dblock	struct	Members: 4-byte d_id, 1-byte d_type, HashTable dblock_data, unsigned char *data
dblock_store	Linked list	Element: Dblock
restore_cache	Linked list	Element: ⟨ 4-byte dblock_id, 4MB dblock ⟩
recipe entry	struct	a 4-byte dblock_id, 1-byte chunk_type, 20-byte fingerprint, 8-byte location
recipe	Linked list	Element: recipe entry

For retrieval, SnapStore creates a recipe file for each function snapshot, which stores the information required to reassemble the data chunks. To minimize disk accesses while reassembling a function snapshot, SnapStore uses per-memory region restore caches that store the data of recently accessed dblocks (during the retrieval process).

4.4 Preprocess Function Snapshot

A serverless platform typically takes a snapshot of the microVM after the function's initialization. SnapStore creates a custom function that first runs the serverless function, and then invokes a custom code snippet before creating the snapshot. The code snippet captures the following information: status (dirty or non-dirty) and type (NRT, RX_MAP, and W_MAP) of a memory page. To identify the status, we keep track of the dirty pages by using the KVM_MEM_LOG_DIRTY_PAGES flag while registering the microVM memory with KVM. To identify the type, we parse the *pagemap* file of the runtime process in the *proc* filesystem. Subsequently, we extract the permissions of the runtime memory pages by parsing the runtime process's *maps* file in the *proc* filesystem. We utilize this information by creating the *reg_info* data structure that stores the mapping between a memory region and its type. As demonstrated earlier in Section 3.2, storing non-dirty memory pages leads to space wastage; we store the function snapshot as a sparse file (linked list like structure for dirty memory regions) by eliminating non-dirty memory pages (discussed in Section 3.2).

4.5 Region-based Backup

In Algorithm 1, we show the steps taken by SnapStore to store a snapshot of a function on a disk. Note that we present a single-threaded view of our multi-threaded implementation, which uses a lock-synchronized queue for inter-thread communication. The lock-synchronized queue guarantees the sequential processing of chunks.

In addition, the operations are atomic to ensure crash consistency. Let us elaborate.

Algorithm 1 Deduplication algorithm

```
 1: procedure CREATE_SNAPSHOT($file$, $reg\_info$)
 2:     Create a recipe file $r$
 3:     $chunks \leftarrow$ REGION_CHUNK($file$, $reg\_info$)
 4:     for all $c \in chunks$ do
 5:         $c$.fp ← SHA-1($c$.data, $c$.len)
 6:         if $c \in$ W_MAP then
 7:             SAVE_CHUNK($c$)
 8:             $re \leftarrow$ GEN_RECIPE_ENTRY($c$) ▷ Generate recipe entry for a chunk
 9:             $r$.insert($re$)
10:             continue
11:         end if
12:         $unique \leftarrow$ IDENTIFY_DUPLICATE($c$)
13:         if $unique \neq NULL$ then
14:             $c$.d_id ← $unique$
15:         else
16:             SAVE_CHUNK($c$)
17:         end if
18:         Update region-specific data structures.
19:         $re \leftarrow$ GEN_RECIPE_ENTRY($c$)
20:         $r$.insert($re$)
21:     end for
22:     Store necessary data structures and $r$ on the disk
23: end procedure
```

Using reg_info, we first split a snapshot of a function into memory regions. Subsequently, we divide the memory regions into k-byte chunks using the fixed-size chunking algorithm (see Line 3). This is because the computational cost of the fixed-size chunking algorithm is lower than that of variable-sized chunking algorithms. Recall that the chunk size of a memory region significantly impacts the data redundancy metric and the fingerprint/index table size (see Section 3.4). Therefore, we select different values of the chunk size for different memory regions. In the case of NRT, we chose the 64-byte chunk size because it provides higher data redundancy (see Figure 4), even though the index table size is large. To compensate for the costly index table size, we chose the 4096-byte chunk size for RX_MAP because the reduction in the chunk size has a negligible impact on data redundancy. We then compute the chunk's fingerprint using the SHA-1 hashing algorithm to identify duplicate data chunks (see Line 5).

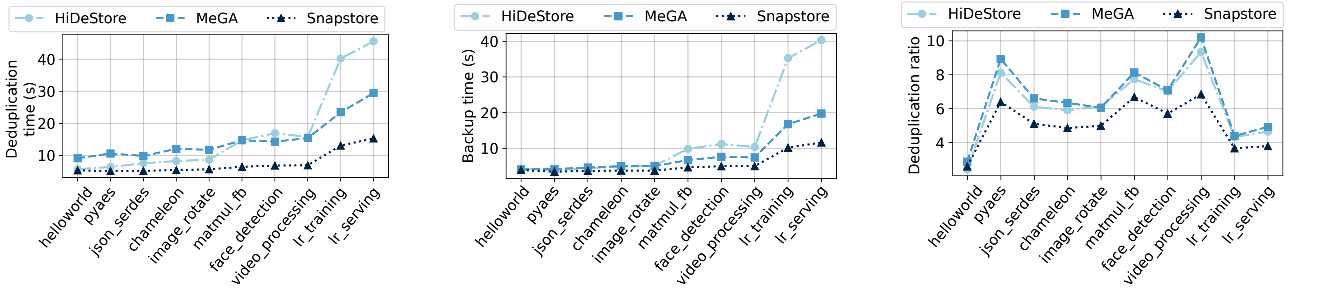
SnapStore utilizes a region-based deduplication scheme, in which we find unique data chunks exclusive to a particular type of memory region. Therefore, we maintain a per-memory region fingerprint cache, index table, dblocks, and dblock store. We do not consider the W_MAP memory region for deduplication because the probability of overlaps is small; hence, it is stored directly in the dblocks (see Line 7). To identify duplicates within the NRT and RX_MAP memory regions, we first perform a lookup operation on the region-specific fingerprint cache, and if that operation fails, we perform the same on the region-specific index table (see Line 12). In case there is a hit, we set the *d_id* member to the value provided by the index table or cache. Otherwise, we store the data chunk in the region's dblock (see Line 16). Subsequently, we add the unique data chunk to the fingerprint cache and index table (see Line 18). If the dblock is full, then we add it to the respective dblock store. We create a recipe entry for every data chunk that contains the necessary information to fetch its data from the dblock store (see Line 19). Finally, we construct a recipe file for a function snapshot by adding recipe entries (see Line 20). We store the ⟨ dblock store, recipe ⟩ on disk – this will be utilized by the retrieval stage to reassemble the snapshot of a function (see Line 22). Note that the deduplication mechanism is immune to address space layout randomization (ASLR). This is because, any snapshot creation always captures the current address layout. The same layout can be seamlessly restored later on (the MicroVM ensures that all starting address pointers are correctly set).

Algorithm 2 Retrieval algorithm

```
 1: procedure RETRIEVE_SNAPSHOT($recipe$ r)
 2:     for all $re \in r$ do
 3:         $c \leftarrow$ BUILD_CHUNK(re)
 4:         if $c \in$ W_MAP then
 5:             $c$.data ← READ_DATA($c$.d_id, NULL)
 6:             WRITE_DATA($c$)
 7:             continue
 8:         end if
 9:         buff ← get_region_buff($c$.c_type)  ▷ Get region specific read buffer
10:         cache ← get_restore_cache($c$.c_type)  ▷ Get region specific restore cache
11:         buff.insert($c$)
12:         if full(buff) then
13:             Sort buff by the d_id attribute
14:             for all $c' \in$ buff do
15:                 $c'$.data ← READ_DATA($c'$.d_id, cache)
16:                 WRITE_DATA($c'$)
17:             end for
18:         end if
19:     end for
20: end procedure
```

4.6 Region-based Retrieval

In Algorithm 2, we show the steps taken by SnapStore to retrieve a function snapshot from the disk (single-threaded view). In our multi-threaded implementation, three threads perform the following tasks: ① reads the recipe entries, ② reads the data from the dblocks, and ③ writes the data into the function's snapshot file located in the RAM-based cache. The retrieval stage first reads the recipe entries from the recipe file. For each recipe entry, it creates a chunk structure (see Line 3). Subsequently, it loads the chunk's data by reading the dblock from the dblock store, whose id and type are equal to the chunk's *d_id* and *c_type* members. The chunk's data is then written into the cache to reassemble the function snapshot file. Note that we need to perform disk read operations for every chunk – this results in a performance bottleneck.



(a) Deduplication time on HDDs

(b) Deduplication time on SSDs

(c) Deduplication ratio

Figure 6: The deduplication performance of SnapStore where function snapshots are stored in *Config-1*. *Note: SnapStore improves the deduplication time by 50% on HDDs and 30.8% on SSDs, while limiting the deduplication ratio by 21.6%.*

To achieve near-sequential disk read operations, we must read data chunks in an increasing order of d_id and d_type. If we attempt to rearrange the recipes in either the retrieval stage or backup stage, then a recipe with a very large number of chunks may still prove to be a bottleneck. Recall that W_MAP dblocks store the original data, whereas NRT and RX_MAP dblocks store unique data chunks. To ensure a sequential disk access pattern, if we encounter a W_MAP type chunk, then we immediately restore (see Line 5-6). Otherwise, we push the chunk to its corresponding memory region's read buffer, whose size is set to the dblock size (see Line 11). If the read buffer is full, we sort the entries of the segment in increasing order of d_id followed by reading them (see Line 13-15). Furthermore, SnapStore uses memory region-specific restore caches to store recently accessed dblocks (see Line 15). Finally, the data chunks are written into the cache (see Line 6,16).

Optimization: To further speedup the retrieval process, we utilize the previously retrieved snapshots on the cache to minimize the number of disk read operations. If a unique data chunk exists in the cache, then we read the chunk directly from the cache rather than the disk. To support this optimization, we need to make a separate recipe file for a function for entries stored in the RAM-based cache.

5 EVALUATION

In this section, we discuss the efficacy of SnapStore for storing and retrieving snapshots. We evaluate SnapStore by comparing its deduplication, and retrieval performance against two state-of-the-art schemes: HiDeStore [30] and MeGA [51]. To demonstrate the *end-to-end* performance, we measure the end-to-end latency of a serverless function when SnapStore is integrated with the FaaSnap serverless platform [3]. Note that whenever we use any comparative term like *increases*, *decreases* or *reduces*, it means that the reported value is being compared with the best value produced by any one of these two state-of-the-art schemes (MeGA or HiDeStore).

5.1 Experimental Setup

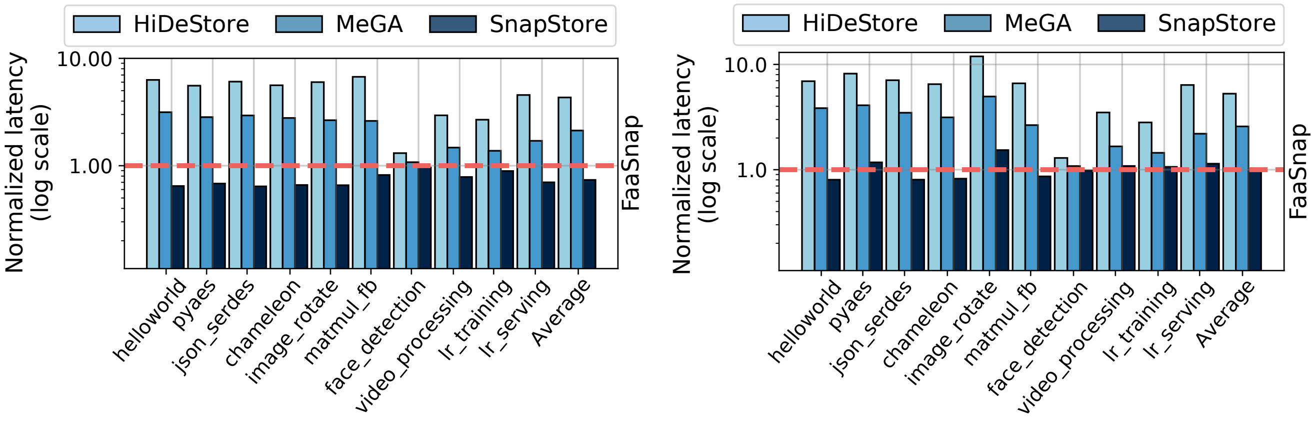
Our evaluation setup has already been described in Table 2 (in Section 3). The per-region restore and fingerprint caches contain 1024 and 4096 entries (similar to HiDeStore [30]), respectively. We use the least recently used (LRU) replacement policy for managing the caches. Prior work evaluated the effectiveness of their schemes by storing and retrieving datasets containing different backup versions of an application [30, 51]. This is because the deduplication and retrieval performance of a scheme depend on the order of backup files. However, to the best of our knowledge, no such backup datasets exist for serverless computing. For a level-playing field, we select the function snapshots in increasing order of their dirty memory size as our backup dataset (similar to QuickDedup [40]). This is the closest that we can get to the setups in which our competing designs were originally evaluated in. Let us call this *Config-1*. Later on, to show the robustness of our proposal, we shall evaluate SnapStore on 70 randomly generated function snapshot sequences (this is enough to achieve a steady state). Let us call this *Config-2*.

For a given sequence of microVM snapshots, we shall store in the given order and then retrieve in the reverse order – this is the standard evaluation methodology, which has also been followed by our competing proposals. We use a 64-byte chunk size and the fixed-size chunking algorithm for both the approaches. To properly capture the disk overhead, we clear the page, dentry, and inode caches prior to deduplication.

5.2 Deduplication Performance

SnapStore performs snapshot pre-processing before deduplication, which can take between 20 and 110 milliseconds (off the critical path). To measure the deduplication performance, we use the following metrics: the deduplication time and deduplication ratio. Given a file, the deduplication time captures the time taken to remove duplicate data chunks and store the unique data chunks on the disk. The deduplication ratio is the ratio of the size of the original data to the size of the deduplicated data. The higher it is, the better it is (≥ 1).

Let us first consider *Config-1*. In Figure 6, we show that SnapStore reduces the deduplication time of a function snapshot by 50% on HDDs and 30.8% on SSDs, on average, when compared to the state-of-the-art approaches, respectively. This is because it omits the memory region with the least probability of overlap during deduplication, i.e., W_MAP. Furthermore, the chunk size for each memory region is chosen wisely. As a result, the number of hash comparisons and computations was reduced by an average of 79.4%, compared to other approaches. MeGA rearranges the unique data chunks of a function snapshot post backup to improve the physical proximity of the latest version in its data containers (dblocks in our design). Whereas, HiDeStore updates its index table so that



(a) End-to-end latency on HDDs (normalized to vanilla FaaSnap)

(b) End-to-end latency on SSDs (normalized to vanilla FaaSnap)

Figure 7: The end-to-end performance of FaaSnap along with SnapStore, MeGA and HiDeStore, where function snapshots are retrieved as per *Config-1* (normalized to that without deduplication). *Note: FaaSnap + SnapStore improves the latency by 25.9% over FaaSnap on HDDs, while it degrades by a mere 0.6% on SSDs, on average.*

it only contains fingerprints of unique data chunks corresponding to the latest version. As a result, these schemes exhibit a high deduplication time as compared to SnapStore.

SnapStore reduces the deduplication ratio by an average of 21.6% compared to the state-of-the-art approaches (see Figure 6c). This is because SnapStore does not consider the W_MAP memory region for deduplication. The deduplication ratio is the highest for MeGA because it further eliminates similarities between the unique data chunks along with duplicate data chunks by using base-delta compression.

Now, let us consider *Config-2*. To study the effect of the order of function snapshots on the deduplication performance, we perform deduplication on 70 randomly chosen sequences of function snapshots. The average improvement in the deduplication time is 46% on HDDs, 36.2% on SSDs, 42.7% on NVMe, and ≈ 60% on a NAS server as compared to the nearest competitor (mostly MeGA). The standard deviation of the improvement ratio for all the storage technologies is limited to 3.3%. With SnapStore the average degradation of the deduplication ratio (with respect to the best performer, MeGA) is 10% (standard deviation ≈ 3.6%).

5.3 End-to-End Performance

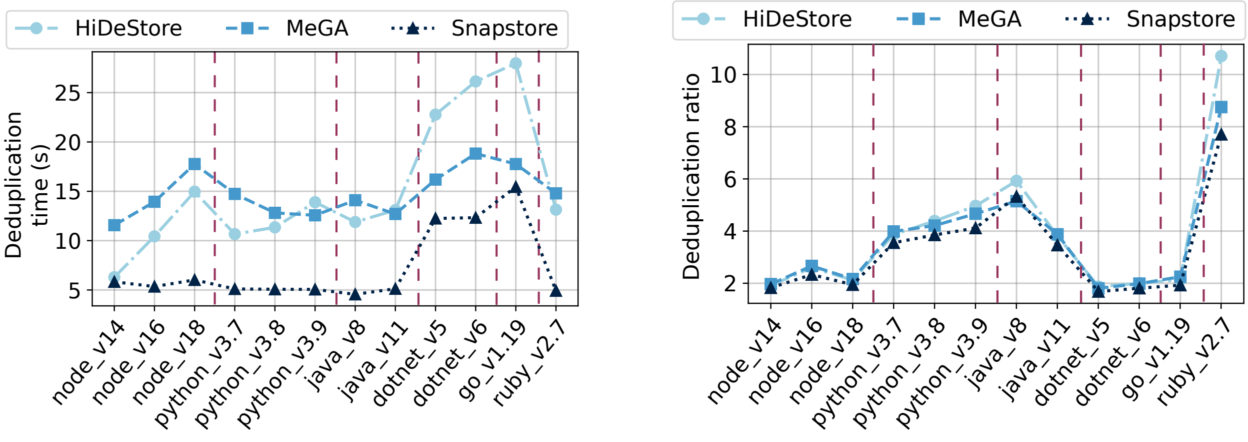
To study the end-to-end performance, we integrate SnapStore and the other two approaches with the FaaSnap serverless platform. To measure the end-to-end latency, we execute a function from a deduplicated snapshot, and add the retrieval time of the function snapshot to the function's execution latency. In Figure 7, we show that SnapStore improves the mean end-to-end latency by 25.9% on HDDs, while degrading the latency by 0.6% on SSDs when compared to FaaSnap. In the case of HDDs, we read 2.4× less data from the storage device and reassemble the snapshot in-memory, which leads to the speedup. On the other hand, the lazy loading of pages from the SSD disk is faster. Therefore, FaaSnap performs marginally better. Moreover, we observe similar results when SnapStore is integrated with other serverless platforms such as vHive [47].

The retrieval time for the other two state-of-the-art approaches increases by 4.2× on HDDs, and 4.3× on SSDs when compared to SnapStore, respectively. This is because SnapStore reassembles the deduplicated function snapshot in the RAM-based cache instead of the storage device. Furthermore, SnapStore retrieves unique data chunks from the cache subject to availability, which reduces the number of dblock accesses by 65.5% compared to retrieving chunks exclusively from the storage device. If we reassemble the snapshot on the storage device instead of the cache, the retrieval time decreases by 30% with HDDs and about 3% with SSDs and NVMes.

In *Config-2*, we get similar results: the average improvement in the retrieval time is 82.6% on HDDs, 75.8% on SSDs, 82.54% on NVMes, and 80% on NAS servers, with a standard deviation of up to 1.76%. In the case of deploying the vanilla serverless platform on ZFS, ZFS takes 1.75× more memory and has a 1.18× higher end-to-end latency to execute a function as compared to SnapStore on HDDs, respectively. SnapStore consumes 280 MB of memory for the deduplication operations, on average.

5.4 Limit Study with Different Runtime Environments

Next, we did a limit study to investigate the efficacy of SnapStore across a set of different runtime environments – each subset of environments had a similar software setup (the version numbers of the frameworks were slightly different). We stored function snapshots of a simple "Hello World" function written using 12 different runtime environments that are supported by Amazon Lambda [27] on HDDs. They are listed in Table 4 – there are minor variations in the version numbers of the same runtime frameworks (for each individual subset). The memory size of the function is set to the minimum possible size, which is 128 MB regardless of its runtime environment. In Figure 8a, we show that the deduplication time of a function snapshot improves by at least 51% because of the memory-region aware deduplication scheme (as compared to MeGA). Moreover, the



(a) Deduplication time **(b) Deduplication ratio**

Figure 8: The deduplication performance of SnapStore when storing function snapshots of different runtime environments in *Config-1. Note: SnapStore improves the deduplication time of a function snapshot by 51%, while limiting the deduplication ratio by 11.9%.*

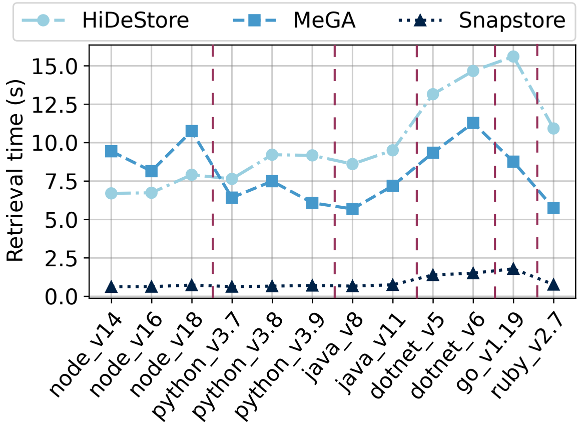


Figure 9: The retrieval performance of SnapStore when storing function snapshots of different runtimes in *Config-1. Note: SnapStore improves the retrieval time of a function snapshot by 88.71%.*

Table 4: List of runtime environments (adapted from Amazon [27]). *Note: The size of the microVM allocated for all the runtime environments is 128 MB.*

Runtime	Version	Runtime	Version	Runtime	Version
Node.js	v14	Node.js	v16	Node.js	v18
Python	v3.7	Python	v3.8	Python	v3.9
Java	v8	Java	v11	.NET	v5
.NET	v6	go	v1.19	ruby	v2.7

number of hash computations and comparison operations reduces by 63.8%. The deduplication ratio decreases by 11.9%, because we omit the W_MAP memory region in the deduplication process (see Figure 8b). The retrieval time of a function snapshot improves by 88.71% (see Figure 9).

In a subset (like all Python or all dot-net), we see an improvement in the dedup ratio, with version changes, mainly because we can reuse the dblocks stored by previous versions. The ratios are also almost the same for all frameworks and we are much better in terms of snapshot/retrieval time. Java-based frameworks are the only exception because there are significant changes across versions [45]. We observe that the deduplication ratio of the Ruby runtime is the highest. This is because 10% of the function’s snapshot is occupied by the runtime process. Let us explain the spikes. An example is the addition of Python 3.7. We see a spike here because the number of dirty-memory pages is reduced by 37% as compared to that in Node 18. To understand the benefits of reading unique data chunks from the cache instead of the disk, we retrieve node_v16 from the HDD, when node_v14 was present in the cache. We observe a performance benefit of 49.8% (as compared to retrieving the entire snapshot from the HDD).

6 RELATED WORK

6.1 Deduplication in Cloud Environments: Overall Operation

To improve space utilization in large cloud computing setups, prior work [19, 21, 40, 50] has proposed using deduplication techniques to eliminate redundancy across backup cloud data, VMIs (VM images) and containers. There exist file systems such as Opendedup and ZFS, that support inline deduplication (eliminating redundancies as it is written into the disk). However, these file systems have high memory requirements [40]. In the case of VMIs, prior work [19, 21] divided a VMI into chunks using fixed-size chunking, and then they eliminated redundancy by using the chunk’s hash. However, these techniques entail the cost of additional hash calculations and comparisons. Quickdedup [40] proposed a duplicate detection algorithm that splits data chunks into buckets by comparing randomly chosen bytes from the data chunks. Subsequently, they perform hashing to identify duplicate chunks in a bucket.

In the case of cloud data, prior work [30, 51] has used either a chunk-level approach or an even finer-grained approach. In the fine-grained approach, in addition to eliminating duplicate chunks, any similarities between the unique chunks are identified and base-delta compression is used to compress the chunk data. Here, the overheads are high.

6.2 Specific Schemes

6.2.1 Duplicate Identification Schemes. Sparse Indexing [33] relies on a sampling technique to determine which fingerprints need to be cached. Extreme Binning [6], on the other hand, minimizes fingerprint cache lookup operations by employing a novel file similarity technique. If a match is found, the remaining fingerprints are retrieved from the disk, and the data chunks are compared. Silo [48] utilizes both the order of chunks and the similarity to expedite the removal of duplicate data. iDedup [44] uses locality in the disk access patterns to minimize disk I/Os and seeks during inline deduplication, thereby reducing the deduplication cost. However, for deduplication it requires a dedicated in-memory cache to store metadata information and a fingerprint table (250 MB to 1 GB) – this leads to a much larger memory overhead vis-a-vis SnapStore (we need 280 MB on an average). The focus of this whole line of work is not on minimizing the retrieval time but on reducing the deduplication cost. We on the other hand focus a lot on minimizing the retrieval time, because it is on the critical path.

HiDestore [30] relies on the simple insight that a given chunk is the most similar to its immediately previous version. Therefore, the fingerprints of the current version are compared with the fingerprints of the previous version, thus minimizing the overheads. On the other hand, MeGA [51] removes the similarity between unique chunks (using base-delta compression), only if the base chunk is contained within a dblock that stores more than a specified number of unique base chunks. The idea is to reuse such dblocks frequently.

These approaches compete with our method, where we use a different method of deduplication based on memory regions. We have compared them with our design in Section 5.

6.2.2 File Restoration Schemes. Prior work used a restore cache and sometimes chunk rewriting techniques to minimize file restoration costs (discussed in Section 2.3). This is because the unique data chunks of a backup stream are scattered across dblocks (referred to as *chunk fragmentation*). We can have two types of restore caches: chunk-level [31] and dblock-level [16, 36]. A chunk-level cache stores the most recently accessed chunk's data in memory. A dblock-level cache, in contrast, stores the most recently accessed dblock's data in memory.

Nam et al. [35] estimated the degree of chunk fragmentation using the following attributes: the optimal number of dblocks required to store a file and the actual number of dblocks utilized to store it. If the metric is below a specified threshold, duplicate data chunks are rewritten to a new dblock. To avoid creating sparsely populated dblocks, Kaczmarczyk et al. [23] rewrite a segment of a file in a new dblock if the overlap between the unique data chunks of a segment and other chunks in the dblock is low. Fu et al. [15] proposed the HAR algorithm, which utilizes historical information to identify sparse dblocks – they are fit candidates for rewriting. Capping [32] techniques, on the other hand, rank dblocks according to the number of unique data chunks of a file they contain. Subsequently, it rewrites the unique data chunks if they are not part of the top k dblocks.

HiDeStore [30] and MeGA [51] focus on the layout of dblocks. The former maintains two types of dblocks: archival and active. Active dblocks store the chunk data of the latest version, and archival dblocks store data for previous versions. MeGA [51] maintains a base-delta compression friendly data layout, ensuring that the unique base data chunks and delta values of the latest version are stored contiguously, preferably in the same dblock.

In SnapStore, there is no need to do rewriting because we ensure near-sequential disk reads owing to our unique data layout (memory region-wise). Furthermore, our scheme is not for backup data and thus all the work on optimizing storage for different versions is not relevant for us.

7 CONCLUSION

This paper shows that our proposed paradigm, which is to attack the problem by treating different memory regions differently can provide good solutions. Our deduplication time and retrieval time gains of nearly 46% and 82.6%, respectively on HDDs, over the state-of-the-art are quite significant and also very relevant in the context of serverless applications. There is, of course, a small decrease in the deduplication ratio ($\approx$ 10%); however, we would like to claim that the performance gains overshadow this. Upon integration with FaaSnap, SnapStore improves the end-to-end latency of functions by 25.9%, while minimizing store space by 2.4× on HDDs. We believe that in the future, it will be possible to write dedicated compilers and libraries that will be able to nicely split code and data into different pre-specified regions and then dedicated hardware or software modules could exclusively focus on those regions that are expected to have a high probability of overlaps with similar regions in other snapshots. This region-based paradigm for serverless processes can perhaps also be used to enhance their security and provide different degrees of persistence.

ACKNOWLEDGMENTS

This work has been supported by the Prime Minister's Research Fellows (PMRF) scheme.

REFERENCES

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

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

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

[4] Microsoft Azure. 2023. *Azure Functions – Serverless Functions in Computing*. Retrieved May 25, 2023 from https://azure.microsoft.com/en-us/products/functions

[5] Jeff Barr. 2022. *Accelerate Your Lambda Functions with Lambda SnapStart.* Retrieved May 25, 2023 from https://aws.amazon.com/blogs/aws/new-accelerate-your-lambda-functions-with-lambda-snapstart/

[6] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. 2009. Extreme Binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.* 1–9. https://doi.org/10.1109/MASCOT.2009.5366623

[7] Gary Bradski. 2000. The openCV library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer* 25, 11 (2000), 120–123.

[8] Zhichao Cao, Hao Wen, Fenggang Wu, and David H.C. Du. 2018. ALACC: Accelerating Restore Performance of Data Deduplication Systems Using Adaptive Look-Ahead Window Assisted Chunk Caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18).* USENIX Association, Oakland, CA, 309–324. https://www.usenix.org/conference/fast18/presentation/cao

[9] Google Cloud. 2023. *Cloud Functions.* Retrieved May 25, 2023 from https://cloud.google.com/functions

[10] IBM Cloud. 2023. *IBM Cloud Functions.* Retrieved May 25, 2023 from https://cloud.ibm.com/functions

[11] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21).* Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3464298.3476133

[12] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20).* Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[13] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2021. Serverless Applications: Why, When, and How? *IEEE Software* 38, 1 (2021), 32–39. https://doi.org/10.1109/MS.2020.3023302

[14] Kave Eshghi and Hsiu Khuern Tang. 2005. *A framework for analyzing and improving content-based chunking algorithms.* Technical Report.

[15] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. 2016. Reducing Fragmentation for In-line Deduplication Backup Storage via Exploiting Backup History and Cache Knowledge. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (2016), 855–868. https://doi.org/10.1109/TPDS.2015.2410781

[16] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. 2014. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14).* USENIX Association, Philadelphia, PA, 181–192. https://www.usenix.org/conference/atc14/technical-sessions/presentation/fu_min

[17] Tal Hoffman. 2021. *Firecracker internals: a deep dive inside the technology powering AWS Lambda · Tal Hoffman.* Retrieved May 20, 2023 from https://www.talhoffman.com/2021/07/18/firecracker-internals/

[18] IBM. 2019. *About Sparse Files.* Retrieved Feb 21, 2023 from https://www.ibm.com/support/pages/about-sparse-files

[19] K. R. Jayaram, Chunyi Peng, Zhe Zhang, Minkyong Kim, Han Chen, and Hui Lei. 2011. An Empirical Analysis of Similarity in Virtual Machine Images. In *Proceedings of the Middleware 2011 Industry Track Workshop* (Lisbon, Portugal) *(Middleware '11).* Association for Computing Machinery, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/2090181.2090187

[20] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21).* Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[21] Keren Jin and Ethan L. Miller. 2009. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (Haifa, Israel) *(SYSTOR '09).* Association for Computing Machinery, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/1534530.1534540

[22] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[23] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. 2012. Reducing Impact of Data Fragmentation Caused by In-Line Deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '12).* Association for Computing Machinery, New York, NY, USA, Article 15, 12 pages. https://doi.org/10.1145/2367589.2367600

[24] Linux Kernel. 2023. *The Definitive KVM (Kernel-based Virtual Machine) API Documentation.* Retrieved May 20, 2023 from https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt

[25] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD).* 502–504. https://doi.org/10.1109/CLOUD.2019.00091

[26] AWS Lambda. 2023. *Lambda execution environments - AWS Lambda.* Retrieved May 19, 2023 from https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html

[27] AWS Lambda. 2023. *Lambda runtimes.* Retrieved Feb 21, 2023 from https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html

[28] AWS Lambda. 2023. *Memory and computing power - AWS Lambda.* Retrieved Feb 21, 2023 from https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html

[29] AWS Lambda. 2023. *Serverless Computing - Amazon Web Services.* Retrieved May 25, 2023 from https://aws.amazon.com/lambda/#:~:text=AWS%20Lambda%20is%20a%20serverless,pay%20for%20what%20you%20use

[30] Pengfei Li, Yu Hua, Qin Cao, and Mingxuan Zhang. 2020. Improving the Restore Performance via Physical-Locality Middleware for Backup Systems. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20).* Association for Computing Machinery, New York, NY, USA, 341–355. https://doi.org/10.1145/3423211.3425691

[31] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13).* USENIX Association, San Jose, CA, 183–197. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lillibridge

[32] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13).* USENIX Association, San Jose, CA, 183–197. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lillibridge

[33] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. 2009. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *7th USENIX Conference on File and Storage Technologies (FAST 09).* USENIX Association, San Francisco, CA. https://www.usenix.org/conference/fast-09/sparse-indexing-large-scale-inline-deduplication-using-sampling-and-locality

[34] Athicha Muthitacharoen, Benjie Chen, and David Mazières. 2001. A Low-Bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP '01).* Association for Computing Machinery, New York, NY, USA, 174–187. https://doi.org/10.1145/502034.502052

[35] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David H. C. Du. 2011. Chunk Fragmentation Level: An Effective Indicator for Read Performance Degradation in Deduplication Storage. In *2011 IEEE International Conference on High Performance Computing and Communications.* 581–586. https://doi.org/10.1109/HPCC.2011.82

[36] Young Jin Nam, Dongchul Park, and David H.C. Du. 2012. Assuring Demanded Read Performance of Data Deduplication Storage with Backup Datasets. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.* 201–208. https://doi.org/10.1109/MASCOTS.2012.32

[37] OpenCV. 2023. *OpenCV: Cascade Classifier.* Retrieved May 25, 2023 from https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html

[38] Abhisek Panda and smruti ranjan sarangi. 2023. FaaSCtrl: A Comprehensive-Latency Controller for Serverless Platforms. (8 2023). https://doi.org/10.36227/techrxiv.24049809.v1

[39] New Relic. 2020. *Executive Summary | For the Love of Serverless.* Retrieved May 25, 2023 from https://newrelic.com/resources/ebooks/serverless-benchmark-report-aws-lambda-2020

[40] Shweta Saharan, Gaurav Somani, Gaurav Gupta, Robin Verma, Manoj Singh Gaur, and Rajkumar Buyya. 2020. QuickDedup: Efficient VM deduplication in cloud computing environments. *J. Parallel and Distrib. Comput.* 139 (2020), 18–31. https://doi.org/10.1016/j.jpdc.2020.01.002

[41] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20).* USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[42] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. 2022. Fireworks: A Fast, Efficient, and Safe Serverless Framework Using VM-Level Post-JIT Snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22).* Association for Computing Machinery, New York, NY, USA, 663–677. https://doi.org/10.1145/3492321.3519581

[43] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3423211.3425682

[44] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. 2012. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast12/idedup-latency-aware-inline-data-deduplication-primary-storage

[45] Daniel Strmecki. 2023. *New Features in Java 11*. Retrieved May 25, 2023 from https://www.baeldung.com/java-11-new-features

[46] Markus Thömmes. 2017. *Squeezing the milliseconds: How to make serverless platforms blazing fast!* Retrieved May 25, 2023 from https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0

[47] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 559–572. https://doi.org/10.1145/3445814.3446714

[48] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 2011. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. USENIX Association, Portland, OR. https://www.usenix.org/conference/usenixatc11/silo-similarity-locality-based-near-exact-deduplication-scheme-low-ram

[49] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 101–114. https://www.usenix.org/conference/atc16/technical-sessions/presentation/xia

[50] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R. Butt. 2020. DupHunter: Flexible High-Performance Deduplication for Docker Registries. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 769–783. https://www.usenix.org/conference/atc20/presentation/zhao

[51] Xiangyu Zou, Wen Xia, Philip Shilane, Haijun Zhang, and Xuan Wang. 2022. Building a High-performance Fine-grained Deduplication Framework for Backup Storage with High Deduplication Ratio. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 19–36. https://www.usenix.org/conference/atc22/presentation/zou