Memory Deduplication for Serverless Computing with Medes

Divyanshu Saxena
The University of Texas at Austin

Tao Ji
The University of Texas at Austin

Arjun Singhvi
University of Wisconsin-Madison

Junaid Khalid
University of Wisconsin-Madison

Aditya Akella
The University of Texas at Austin

Abstract

Serverless platforms today impose rigid trade-offs between resource use and user-perceived performance. Limited controls, provided via toggling sandboxes between warm and cold states and keep-alives, force operators to sacrifice significant resources to achieve good performance. We present a serverless framework, Medes, that breaks the rigid trade-off and allows operators to navigate the trade-off space smoothly. Medes leverages the fact that the warm sandboxes running on serverless platforms have a high fraction of duplication in their memory footprints. We exploit these redundant chunks to develop a new sandbox state, called a dedup state, that is more memory-efficient than the warm state and faster to restore from than the cold state. We develop novel mechanisms to identify memory redundancy at minimal overhead while ensuring that the dedup containers' memory footprint is small. Finally, we develop a simple sandbox management policy that exposes a narrow, intuitive interface for operators to trade-off performance for memory by jointly controlling warm and dedup sandboxes. Detailed experiments with a prototype using real-world serverless workloads demonstrate that Medes can provide up to 1×-2.75× improvements in the end-to-end latencies. The benefits of Medes are enhanced in memory pressure situations, where Medes can provide up to 3.8× improvements in end-to-end latencies. Medes achieves this by reducing the number of cold starts incurred by 10-50% against the state-of-the-art baselines.

CCS Concepts: • **Networks** → **Cloud computing**; • **Information systems** → **Computing platforms**; *Data centers*; • **Software and its engineering** → *n-tier architectures*.

Keywords: Serverless, Memory Deduplication, Cloud Computing, Virtualization

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3524272

ACM Reference Format:
Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory Deduplication for Serverless Computing with Medes. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3492321.3524272

1 Introduction

In the serverless computing paradigm, developers submit a piece of code (*function*) to the serverless platform. A function instance is invoked based on a developer provided-trigger and launched to execute in a sandbox (e.g., a container) with the needed libraries and dependencies loaded. The platform scales the number of functions instances (up or down) based on the number of function invocations per unit time. Serverless computing has become quite popular because (1) it enables application developers to focus on application logic by shifting the burden of provisioning, managing, and scaling resources onto the cloud providers, and (2) it offers cost-efficiency via fine-grained billing where developers only pay for the time when their functions were actually running.

With more demanding applications and workloads migrating to serverless platforms, a key question for providers is how to meet tight performance requirements while also ensuring resource efficiency. Performance heavily depends on how quickly a function instance can start acting on an end-user request. Resource efficiency is achieved by closely matching actively operating sandboxes to the incoming demand.

Serverless platforms today manage performance and efficiency by toggling sandboxes between two states: cold and warm (or paused). A "cold" sandbox does not use any memory resources but induces long cold startup delays (which could be in the order of seconds, depending on the platform and runtime [38]) due to the needs for the execution environment of a function to be initialized and loaded before a function instance can execute in the sandbox [10, 11, 13, 16, 26, 29, 38]. "Warm" or paused sandboxes are kept in memory for a certain amount of time – the *keep-alive period* [29, 38] – after the completion of function execution, consuming significant memory. However, they enable sandbox reuse – subsequent function invocations that arrive before the keep-alive expiry run on these sandboxes experiencing a "warm startup" which

is significantly smaller (varying from 1ms to 20ms depending on the runtimes [38]).

Unfortunately, today's platforms induce highly constraining trade-offs between performance and efficiency, with good performance achievable only at significant resource expense, and make it difficult for operators to control such trade-offs, i.e., tune the achieved performance by controlling resource use. In this paper, we present a new mechanism that increases the flexibility of the trade-off space, enabling better performance (efficiency) for the same efficiency (performance) as platforms today, and a simple way for operators to navigate the trade-off space.

Prior works have attempted to improve trade-offs in a few different ways, but they are either ineffective in practice or do not offer enough flexibility to navigate the trade-offs. For example, techniques such as eschewing fixed keep-alives (used today [16, 29]) in favor of adaptive [29] keep-alive policies, or provisioning sandbox resources in anticipation of future invocations [6, 30] fall short because the unpredictable nature of serverless workloads makes it very difficult to design a general keep alive or pre-warming policy (Section 7.5). Recent proposals [10, 11, 13] to reduce startup times offer cold starts that are still orders of magnitude slower than warm starts or sacrifice code compatibility/isolation, which precludes their widespread adoption [8] (Section 8).

Our work improves the trade-off space today by introducing a new sandbox state with a memory footprint and startup performance in between those of cold and warm states. The third state is built on extending the "reusable sandbox" construct that underlies the warm state today to that of a *reusable sandbox chunk* (RSC). An RSC corresponds to any memory chunk of warm sandboxes that can be "re-used" by other sandboxes. Our empirical study (Section 2) shows the promise of RSCs – we find evidence of significant duplication in the memory states of warm sandboxes; specifically, we find that (1) sandboxes of the same function can have upto 85% duplication in their memory state; (2) even across sandboxes of different functions, we can identify upto 80-90% duplication. Essentially, the RSC notion works by removing such redundant memory chunks across sandboxes, thereby significantly improving serverless platforms' memory-performance trade-offs.

The new sandbox state that RSCs help us introduce is called the *deduplicated* state (or dedup for short). In this state, all the redundant memory chunks of the sandbox are "removed" and only "unique" chunks are stored in memory. Specifically: (1) we store only one copy of an RSC in a "base" sandbox, and dedup-ed sandboxes' memory contents exist as a collection of local completely-unique chunks and redundant RSCs in multiple remote base sandboxes; (2) prior to launching a function, we restore a dedup-ed sandbox by putting together unique local chunks with redundant RSCs read over the network from remote base sandboxes.

Our deduplication approach ensures that the dedup state has a significantly smaller memory footprint than warm startups and that dedup startups are significantly faster than cold starts. By deduplicating more sandboxes, the overall memory usage is smaller than that of a platform which only has warm sandboxes. Further, we can utilize this saved memory to keep more sandboxes, leading to improved performance. Thus, we can leverage the dedup state to improve the flexibility and scope of the memory-performance trade-off in serverless computing.

We present Medes (**Me**mory **De**duplication for **S**erverless), a novel serverless framework that incorporates the dedup sandbox state. Medes makes use of a novel deduplication mechanism that can identify potentially similar chunks in the memory states of sandboxes *across the cluster*. Calculating the amount of chunk-level duplication between the memory states of two sandboxes, and exploiting said redundancy is not trivial on a serverless platform, where a large number of sandboxes could be in memory at the same time, each having tens of thousands of pages in its memory state. Medes tackles this by using three techniques (details in Section 4). First, to ensure scaling and to lower the computational costs of deduplication, while we identify redundancy at the chunk-level (because it is the most effective), we perform deduplication at the *page* granularity: for each page of a dedup sandbox, Medes identifies a similar (base) memory page on the cluster and computes a "patch" with respect to it. The choice of this base page is based on an estimate of the number of RSCs in common between the two pages. Second, to further improve scaling, we restrict the number of base pages to keep track of by demarcating certain sandboxes as base sandboxes and using only the memory pages of these base sandboxes as reference for computing patches. Third, we leverage value-sampled fingerprints [9] to lower the computational and storage costs of redundancy identification - which leads to overall faster function startup times than cold starts and hence promises better function performance.

Medes allows serverless platform operators to control the memory-performance easily trade-offs and navigate the trade-off space via a novel sandbox management policy. Our policy jointly controls the number of warm *and* dedup sandboxes in memory and offers a narrow interface with simple, intuitive parameters through which operators can: (a) reason about the performance achieved for a given resource footprint, (b) control performance (memory) by directly and adjusting memory footprint (performance goals), and (c) customize the policy for different serverless functions.

We evaluate Medes against state-of-the-art keep-alive-based sandbox management policies on real-world serverless workloads. We observe that Medes can deliver on its promise of better performance and efficiency across a spectrum of settings. Medes can provide up to 1×-2.75× smaller function startups in the tail by reducing the number of cold starts incurred by 10-50% against the baselines. Medes achieves this

Benchmark Testcase	Python Libraries	Description
Vanilla	Math, Time	Simple mathematical computation.
LinAlg	Numpy, Time	Linear algebra functions.
ImagePro	Numpy, Pillow	Image processing operations.
VideoPro	Numpy, OpenCV	Video processing functions.
MapReduce	Multiprocessing	Multi-process mapreduce job.
HTTPServe	Chameleon, JSON	Serve an HTML table over HTTP.
AuthEnc	Pyaes, JSON	Encryption/decryption.
FeatureGen	Scikit-learn Tf Idf Vectorizer, Pandas	Data preprocessing and feature generation operations.
ModelServe	Pytorch	Serve an RNN model using serverless functions.
ModelTrain	Scikit-learn Tf Idf Vectorizer and Logistic Regression	Train a classifier.

Table 1. Description of various python libraries in FunctionBench used in our measurement study in Section 2.

by heavily deduplicating warm containers, using which it can keep 7.74-37.7% more sandboxes in memory compared to current state-of-the-art alternatives. Crucially, we observe that the benefits provided by Medes increase under memory pressure, where it provides up to 3.8× improvement in the end-to-end latencies.

2 Background and Motivation

We begin by showing evidence of significant memory redundancy in realistic serverless workloads through a measurement study. We then describe the opportunity that memory redundancy provides to make serverless platforms more performant and memory-efficient.

2.1 Memory Redundancy in Serverless Workloads

Our initial hypothesis is that serverless workloads should exhibit memory redundancy at any given point of time. The intuition is that there are likely multiple warm sandboxes corresponding to the same function in the cluster. Moreover, we expect to see redundancy across different functions as well because different functions can use the same runtime and libraries depending on the use case (see Table 1).

To verify our hypothesis, we compare the memory state of sandboxes corresponding to several real-world serverless functions. We use the FunctionBench [20] suite which consists of python serverless functions corresponding to common use cases such as linear algebra, authentication, HTTP serving, image and data processing, as well as model training and serving. The Python libraries in each use-case are summarized in Table 1. The memory state is obtained by checkpointing sandboxes using CRIU [3] and we initially turn off address space layout randomization (ASLR) to measure the upper bound on redundancy.

To compute the redundancy between two different sandboxes (say A and B), we use the Rabin fingerprinting approach [33], where we sample a chunk of K bytes at regular fixed offsets of $2K$ bytes (the choice of K is discussed later). Then, we compute the SHA1 hash values of sandbox A's chunks at this offset, add them to a hash table, and then check for sandbox B's chunks in the hash table. If there is a match, we check if the actual bytes were the same. Thereafter, we extend both the chunks to include the non-hashed bytes in the memory state to a maximum of $2K$ bytes. We take the maximum common subsequence of bytes from this $2K$ byte sequence. The redundancy of sandbox B with respect to sandbox A is calculated as the percentage of duplicated bytes in sandbox B.

Same Function Sandboxes. We first measure the redundancy present in sandboxes belonging to the same function. Figure 1a demonstrates that there is significant redundancy — as high as 90% — in sandboxes belonging to the same function. Moreover, we see that the amount of redundancy reduces as the chunk size increases. This is because with larger chunk sizes, the probability of one of the bits differing, in the two chunks, increases. In a nutshell, we see that *with a sufficiently fine-grained chunk size, serverless functions exhibit a high degree of redundancy across its sandboxes*. We observe that even after enabling ASLR, we can still identify significant duplication in the memory states (see Figure 1b). This is because we use a high sampling frequency to generate fingerprints and the chunk sizes are smaller than the page level address randomization employed by ASLR. The small drop in duplication (~5% for 64B chunks) is because ASLR employs stack address randomization at the granularity of 16B. However, the sandbox memory state comprises majorly of heap memory, file mappings, and shared libraries, across which redundancy still exists.

Different Function Sandboxes. Next, we measure redundancy across different function sandboxes. To do so, we measure the redundancy of each serverless function in FunctionBench relative to the other serverless functions (using a chunk size of 64B). We see in Figure 1c that there *exists redundancy across sandboxes corresponding to different functions* and the extent depends on the underlying runtime and libraries that are common across the functions. For example, FeatureGen and ModelTrain both use the common module of TfIdfVectorizer. This implies that the entire memory state that the TfIdfVectorizer maintains will likely be largely present in both functions.

Real-world Serverless Workload. Finally, we estimate the amount of memory savings that can be obtained in real-world serverless workloads by leveraging the memory redundancy existing in serverless functions. To do so, we use the various arrival patterns in the serverless production traces (30 min duration) released by Azure [29] and assign them to the use cases in FunctionBench. Figure 2 shows the amount of memory saving that could be achieved in real-world serverless workloads, if we were to tap into the memory redundancy that exists in warm sandboxes. Specifically, we observe that we could get up to 30% memory savings relative to current state-of-the-art platforms that do not leverage the redundancy.

2.2 Deduplicated Sandboxes via RSCs

We view this memory state redundancy among warm sandboxes as an opportunity to overcome the rigid tradeoff imposed by cold and warm starts today. Specifically, we leverage

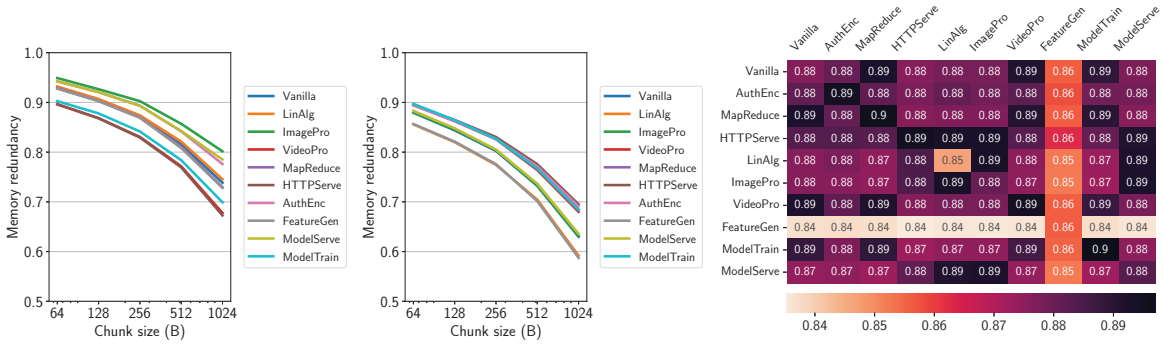

Figure 1. Memory redundancy in serverless workloads. Redundancy between sandboxes of the same function w.r.t. chunk size (a) with ASLR disabled and (b) with ASLR enabled. (c) Cross function redundancy - redundancy of functions on vertical axis w.r.t. those on horizontal axis with 64B chunk size.

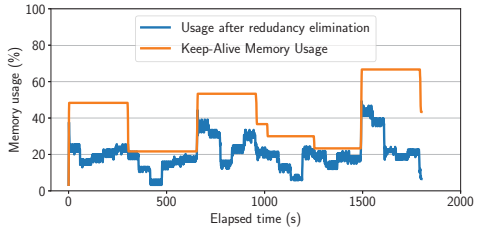

Figure 2. Possible memory savings in real-world serverless workloads by eliminating memory redundancy.

such redundancy by introducing a new sandbox state called the *deduplicated* state (or *dedup* in short), which is a middle ground between cold and warm in that it keeps only part of a sandbox's memory state that is unique in the cluster and eliminates the chunks that exist in a warm sandbox.

For a dedup sandbox to recover, the duplicate chunks are read from the location of the existing copy. In other words, we are *reusing* the existing chunks in warm sandboxes. Therefore, we call them *reusable sandbox chunks* (RSCs). We choose 64 bytes as the size of RSC for the rest of this paper for it yields the highest redundancy, as discussed above. The dedup state, as we will show later, on the one hand, saves memory compared to the warm state - allowing more sandboxes to be kept in memory at the same time, while on the other hand shortens the start time compared to the cold state - reducing the latency of individual function requests.

Section 8 discusses additional recent efforts that strive to improve the performance and efficiency of serverless clusters.

3 Medes Overview

We present Medes, a serverless platform that incorporates the dedup state and provides an easy interface to navigate the trade-off space of performance and memory by jointly controlling warm and dedup sandboxes.

3.1 Architecture

Figure 3 shows the architecture of Medes. Medes consists of a controller and several nodes where functions are executed, interconnected by a cluster/datacenter network.

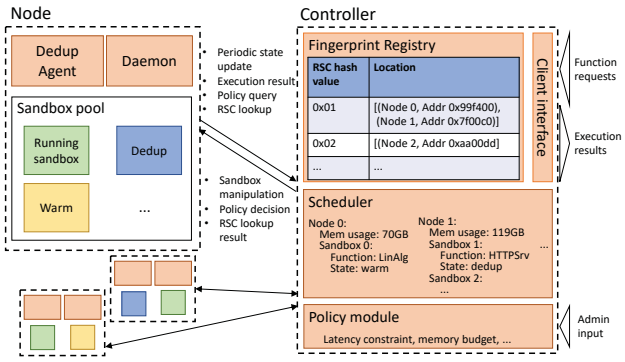

Figure 3. Medes Architecture.

The controller has four major components: 1) the interface to clients, through which function requests are submitted, and results are retrieved; 2) the scheduler that keeps track of the system-wide status (e.g., the resource usage and the warm and dedup sandboxes on each node) spawns a new sandbox or assigns an existing sandbox to serve an incoming request and decides whether to transition a sandbox that has finished to the warm or dedup state; 3) the fingerprint registry, which is a hash table that contains the hash values of RSCs and their corresponding location in the cluster for deduplication; and 4) the policy module that stores policy parameters such as the latency and memory constraints. Specifically, to support deduplication, Medes adds the latter two components to the controller used by serverless platforms today for authentication and dispatching all incoming user requests [4]. Medes takes several critical design decisions to ensure that the deduplication overheads on the controller are minimal, which we discuss in Sections 4.1.3 and 4.2. Additionally, we discuss how the controller can be distributed across multiple nodes to avoid a single node controller from being a scalability bottleneck in Section 4.3.

Each node consists of 1) the daemon that manipulates local sandboxes upon the controller's directives and updates the controller of the node's status; and 2) the dedup agent that performs the deduplication for local sandboxes as indicated by

the controller (via the daemon), and restores local sandboxes from the dedup state when requests are assigned to them.

Next, we provide intuition behind the functionality of these components, outlining their operation at the chunk level and ignoring concerns such as overhead and scale. In Sections 4 and 5, we provide a deep-dive into how Medes actually implements these functionalities and addresses these concerns.

3.2 Basic Workflows at a High-Level

The client submits a function request to the controller's interface in the form of an RPC. The scheduler chooses an available warm or dedup sandbox that can run the function according to its knowledge of the status of the nodes, and hands over the request to the daemon on the chosen sandbox's node. If such a sandbox does not exist, the scheduler by default requests the daemon on the node with the least memory usage to spawn a new one (as long as other resource requirements of the function are met). The daemon prepares the execution environment in case of a new sandbox, while a warm sandbox needs minimal preparation to start. If a dedup sandbox is chosen, the daemon invokes the dedup agent to perform a procedure called the *restore operation*, in which the dedup agent reconstructs the sandbox by reading back a list of RSCs from their locations in the cluster. This list is generated when the sandbox is put into the dedup state.

When a function finishes, the sandbox is moved to the warm state, where the controller may decide whether to dedup the sandbox. The decision is made according to the policy module pre-configured by the administrator (discussed in Section 5). If the controller decides to dedup, the dedup agent invokes a procedure called *dedup operation*. The agent then checks the chunks against the RSC hash values in the fingerprint registry, purges the part of the state that is deemed redundant and records the locations of the RSCs obtained from the fingerprint registry, locally at the dedup agent. We describe Medes's implementation of the dedup and restore operations in greater detail in Section 4.

3.3 Sandbox Lifecycle

As another perspective to Medes' operation, Figures 4a and 4b contrast the sandbox lifecycle state machine on existing platforms and that with the dedup state under Medes. Upon completing execution, the sandbox goes into a warm state in both cases. The sandbox is removed at the expiry of a 'keep-alive' period or if it is evicted in the face of memory pressure (to make room for more sandboxes of other functions). In contrast with today's platforms where a sandbox is purged after a single keep-alive period, Medes allows running a custom policy to determine which state the sandbox is to be transitioned into in order to manage memory and performance. The policy is executed at the global controller to make use of cluster-wide metrics to make decisions. This policy module is invoked periodically by the dedup agent to get decisions for sandboxes. To this end, Medes introduces two knobs in addition to the 'keep-alive' period of traditional keep-alive

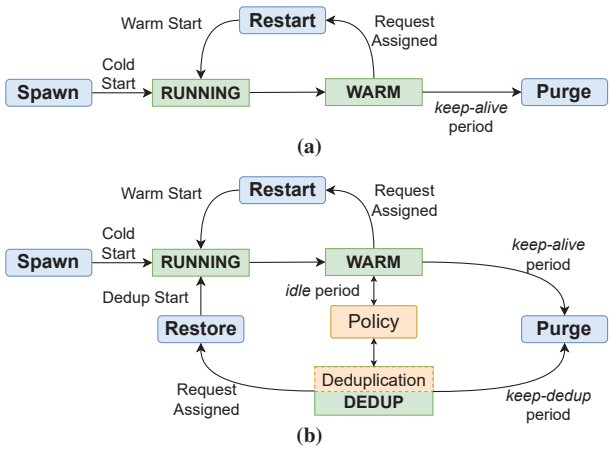

Figure 4. Lifecycle of a sandbox running on (a) Existing Platforms (b) Medes

policies. The first is called the 'idle period'. When a sandbox is in a warm state, upon expiry of this period, the local node's daemon checks with the Medes controller regarding whether to transition the sandbox to a dedup state or keep it warm. The second parameter is the 'keep-dedup period'. When this expires, the local node purges the dedup sandbox from memory. This is similar to the 'keep-alive' period, but separating the two enables Medes to keep dedup sandboxes for a different duration of time, based on the memory-performance trade-off imposed by dedup sandboxes.

4 Medes Dedup and Restore Operations

To extract the complete benefits offered via the dedup sandbox state, the deduplication (dedup) and restoration operations need to be scalable and fast as typically a serverless platform handles execution requests corresponding to different functions (possibly from different tenants) whose load can grow arbitrarily.

Conversion of a warm sandbox to a dedup sandbox, through the deduplication operation consists of the two high-level steps - redundancy identification and redundancy elimination (see Figure 5). The local dedup agent on the machine initiates the memory checkpoint of the warm sandbox, which gives a dump of the corresponding memory state. Given this memory state, the dedup agent identifies duplicate memory chunks by interacting with the controller, which maintains a complete view of the already existing unique memory chunks strewn across the cluster in the form of RSCs. Having identified the redundancies, the dedup agent eliminates (removes) the duplicate memory chunks at the granularity of entire pages and, in doing so, computes a patch for the page being eliminated relative to the base page(s) corresponding to its RSCs. In this manner, Medes reduces the memory footprint of a deduplicated sandbox as it only maintains patches (including any unique leftover pages and memory chunks).

The restore operation converts a deduplicated sandbox to a warm one. This operation involves reading the base pages on which the patches corresponding to this sandbox were calculated and then reconstructing the original pages and recreating

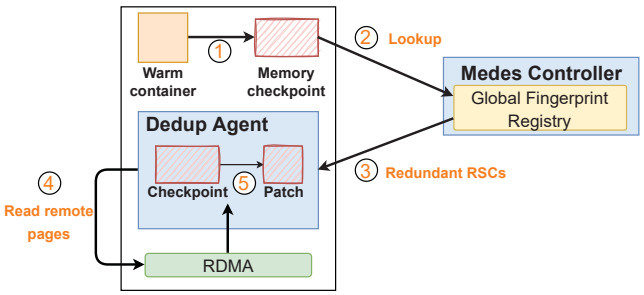

Figure 5. Medes Workflow of the Deduplication Mechanism: 1. Sandbox checkpoint gives the dump of the memory state. 2. RSCs from the memory state are sent to the global hash table on the controller for lookup. 3. The controller sends back the information about redundant RSCs. 4. The Dedup Agent reads all the physical addresses sent by the controller and computes patches of the pages with RSCs. 5. Finally, the memory checkpoint is removed, and only the (smaller) patch is kept in memory.

the memory checkpoint. Finally, the sandbox transitions to the warm state via restoration from this checkpoint.

We now discuss the various design choices we made to ensure that both these operations are fast, scalable, and effective.

4.1 Dedup Op Deep-Dive

We now discuss the granularities of redundancy identification and of redundancy elimination, and the implications for scale, low overhead, and deduplication effectiveness.

4.1.1 Redundancy Identification Granularity.

The redundancy identification granularity, which also corresponds to the size of RSCs, plays a crucial role in identifying duplicate memory chunks. Smaller identification granularities lead to identifying more redundant chunks, but they can lead to hash collisions in the fingerprint registry (as we show in Section 7.8). On the other hand, large identification granularities lead to lesser hash collisions but identify fewer redundant chunks.

In Medes, we balance this trade-off and choose 64B memory chunks as the granularities for identifying redundancies. We empirically observe that this setting enables us to find significant redundant chunks (see Figs. 1a-1c), leading to memory efficiency while having minimal hash collisions.

4.1.2 Redundancy Elimination Granularity.

Once redundant memory chunk identification occurs at the small 64B identification granularity, the next step is to remove the redundant memory chunks. As discussed earlier, the dedup agents need to interact with the controller to get information about the RSCs corresponding to the duplicate memory chunks. A naive approach would eliminate chunks at the same granularity of identification. However, this would imply storing metadata for each small chunk (because we will need to retrieve them during restoration). We observed that sandbox memory states could span up to 100MB on the used benchmarks, which corresponds to nearly ~25K pages, and using all 64B chunks imply metadata for nearly 1.6M chunks for just one sandbox.

With Medes, our key insight is to *decouple* the granularities of redundancy identification and redundancy elimination. To avoid scalability bottlenecks, we default to memory pages as the granularity for redundancy elimination. We now discuss how to identify a redundant page given the RSC information stored in the fingerprint registry.

A strawman approach would be to see if all the 64B memory chunks of the page have their RSCs present in the cluster. However, this would again lead to the communication channel between the dedup agent and controller becoming a bottleneck. Instead, we use *page fingerprints* to reduce the communication overhead and identify the best "base page" corresponding to the page that is under consideration for deduplication.

Page Fingerprints. With Medes, we use a small subset of memory chunks, *value sampled* based on the last two bytes of the chunk; i.e., we conduct a scan of each page over a rolling 64B window and select a 64B chunk as a fingerprint if it is last two bytes match a specific pattern. This approach is straightforward and computationally lightweight as it involves a single linear scan and a lightweight equality check over two bytes.

We use five such value-sampled chunks per page (sensitivity to this evaluated in Section 7.8). This unordered set of five chunk hashes then acts as a fingerprint of the page. The number of overlapping fingerprints between two pages represents the similarity between the two pages. The use of value sampling reduces the communication bottleneck. Still, it delivers on memory efficiency via deduplication as two 'similar' pages typically have a high count of other duplicate memory chunks between them.

Base Page. Given the page fingerprint, each value sampled memory chunk is looked up in the fingerprint registry. For each memory chunk found on the registry, we get a list of candidate pages that have the corresponding RSC. Combining all the candidate pages, we get a candidate set for each memory page. To keep the per-page metadata maintained at the agent small, we choose a single best candidate page from this candidate set - which we call the 'base page' for the respective dedup page. The candidate with the maximum number of duplicate chunks amongst the sampled chunks is chosen to be the base page. If more than one candidate has the same maximum duplication, then the page available locally on the same machine as the page to be deduplicated is chosen.

In this manner, Medes uses the 64B identification granularity to identify a similar 'base' page. Thereafter, a diff or a patch is computed for the deduplicated page against the base page. This patch consists of the unique bytes of the deduplicated page and short metadata information about which range of bytes from the base pages should be appended at what offsets on the patch. Since the base page is likely to be significantly similar to the dedup page, the computed patch is smaller in size than the original page, resulting in a lower memory footprint per page. We use the Xdelta3 [1] library

to compute patches of binary pages. Xdelta3 provides ten compression levels - 0 through 9 (0 indicating no compression while 9 indicating maximum compression), and we use the compression level of 1 to make the restore op fast (see Section 4.2).

4.1.3 Low-footprint Fingerprint Registry.

Until now, we focused on how redundancy identification and elimination occur by leveraging the deduplication information present in the fingerprint registry. We now discuss which sandboxes to use to populate the registry. Inserting memory chunks from all sandboxes would cause the memory footprint of the registry to explode. Given the sampling as mentioned above, we still have nearly ~100K chunks to be stored for each sandbox. In our experiments, we observed that the platform could have thousands of sandboxes at the same point in time. Storing the RSCs from all of these sandboxes can lead to high memory usage.

Hence, we demarcate specific warm sandboxes on the platform as 'base sandboxes'. Only the unique memory chunks of these base sandboxes get inserted into the registry. This design choice is based on the fact that the percentage of memory duplication between any two sandboxes of a given pair of functions $F1$ and $F2$ remains the same. We leverage this property to reduce the amount of warm state by just using the memory chunks of any sandbox (among all sandboxes that ran function $F1$, for example), we can identify the duplicate chunks in all other sandboxes as well. To ensure that the memory state of the base sandbox is not purged, a refcount is maintained by the controller for each base sandbox.

To reduce the impact of the unavailability of a base sandbox, we increment the number of base sandboxes as the number of dedup sandboxes for a function increases. Specifically, we choose a threshold T and demarcate one more base sandbox, when $D/B > T$, where D is the number of dedup sandboxes for a function, and B is the number of base sandboxes. This approach reduces the size of the fingerprint registry to the order of the number of base sandboxes (controlled by the factor T), which is a small number compared to the total number of sandboxes running on the platform.

For our implementation, we use T to be 40. We observe in our evaluation that such an optimization is enough to efficiently fulfill 5X magnified production traces without incurring significant overheads (discussed in Section 7.7). Further, increasing the number of base images may not always help because the different base images are also likely to have a high amount of redundancy, which imposes an unnecessary memory cost.

4.2 Restore Op Deep-Dive

Deduplicated sandboxes would be required to serve requests when all the existing warm sandboxes (if any) corresponding to this function are busy. However, given that we only store patches relative to the base pages for a deduplicated sandbox, such a sandbox needs to be restored before it can serve the incoming request.

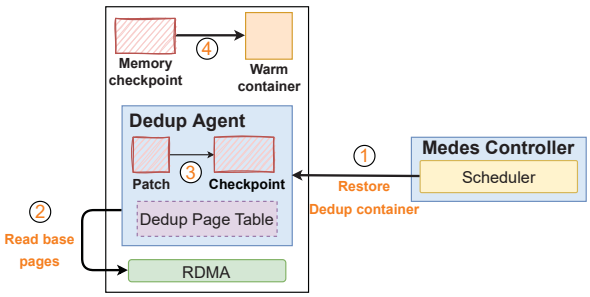

Figure 6. Workflow of the Restoration Mechanism: 1. The scheduler decides when to make a dedup start. 2. The Dedup Agent fetches duplication information about the sandbox from its local data structure. Then, it reads the base pages from the respective nodes. 3. Original (pre-deduplicated) pages of the dedup sandbox are computed using the patch and the base page. These pages are collated to create the memory dump of the sandbox. 4. A container restore mechanism puts the container back into its running state.

The restore operation involves reconstructing the sandbox using the stored patches and the corresponding (possibly remote) base pages. Figure 6 shows a pictorial representation of the restoration procedure. The key challenge is to ensure that the reconstruction and restoration process is fast. In other words, the time taken to reconstruct the memory state of a dedup sandbox should be significantly smaller than the time required to load a new sandbox in memory. Ensuring this makes *dedup starts* significantly faster than cold starts and an intermediate state between warm and cold.

Medes employs three techniques towards this goal. First, dedup starts require restoration of the sandbox, which, apart from restoring the sandbox memory state from the patches and their base pages, involves additional steps such as sandbox namespace creation as well as reconstruction of the process tree (inturn invokes multiple `fork()` system calls) [3]. To speed up the sandbox restoration, Medes performs these additional time-consuming [26] steps prior to deduplicating the sandbox, leaving only memory state restoration during dedup starts. Additionally, Medes saves the container memory checkpoints in-memory to ensure fast restores rather than restoring from disk. In our implementation, these optimizations brought down the time spent to restore the memory state of the dedup-sandbox (before beginning function execution) from 650ms to ~140ms. Secondly, the information required to complete the restoration (e.g., patches and the address of the base page) is stored locally on the machine where the deduped sandbox resides and is managed by the dedup agent. Finally, we leverage the RDMA `read` operation to directly fetch base pages from the remote machine's memory, which avoids the use of remote CPU for communication and (also) yields low latency [40].

4.3 Controller Scalability

Medes adds two components to the serverless controller – fingerprint registry and the policy module – to support deduplication. Through key design decisions of using representative page fingerprints, populating only base sandboxes to the fingerprint registry and avoiding communication with the controller during restores, Medes reduces the overheads at the controller so that it does not become a bottleneck and can cater to workloads of varying scales. To further scale the Medes controller, it can be seamlessly distributed on the same lines as proposed by prior works for centralized serverless controllers [17, 28, 30, 32]. Accesses to the fingerprint registry are independent lookups for each page and the policy module runs the policy on a per-sandbox basis. Therefore, these components can be distributed using conventional techniques for sharding or key-based partitioning [12, 25] along with chain replication [27] (for fault tolerance).

5 Sandbox Management Policy

With Medes, we expose an intuitive interface for the providers to specify the performance and efficiency expectations on a per-function basis. In turn, the platform leverages the ability to deduplicate sandboxes to navigate the memory-performance tradeoff. To this end, we develop a policy that decides whether sandboxes should be kept in the warm state or deduplicated. The policy runs after a warm sandbox does not receive any request for the 'idle period' duration.

Ideally, such a policy should determine the optimal number of warm and dedup sandboxes for any given function, such that it can meet the request arrival rates for that function while efficiently using cluster resources. Hence, the policy must make decisions based on: (i) request arrival rates for the function, (ii) cluster memory pressure, (iii) memory savings because of deduplication, and (iv) overheads of restoring deduplicated sandboxes.

5.1 Dedup and Restore Overhead Considerations

To develop a policy that can make well informed sandbox-granular decisions, we need to account for the fact that the dedup state imposes memory and performance overheads compared to warm starts.

To quantitatively estimate the impact of dedup starts on function performance, we define the *sandbox reuse period* as the minimum time interval between two function invocations on the *same* sandbox. This equals the request execution time *plus* the sandbox startup time. If the sandbox reuse period is R, then in time T, the sandbox can serve a maximum of T/R requests. Compared to warm sandboxes, dedup sandboxes have a higher reuse period (due to the additional time to reconstruct the sandbox checkpoint and restore the sandbox memory state from the checkpoint). They hence can serve fewer requests in a given interval.

Furthermore, the reconstruction of the memory state during the restore op entails additional memory to read the base pages and compute patches. Hence, frequent dedup starts can lead to memory overheads outweighing the memory savings.

5.2 Optimization Problem

The sandbox management policy must decide how many sandboxes must be kept in the dedup state, given a certain load requirement while satisfying a latency bound *and* a memory constraint and accounting for dedup overheads.

5.2.1 Platform Constraints

We denote the current number of sandboxes on the platform by C and the maximum request arrival rate that must be met by λ_{max}. Then, denoting the number of dedup sandboxes be D and the number of warm sandboxes be W, we have

$$W + D = C \tag{1}$$

If C is insufficient to handle the load, the controller spins up additional sandboxes (moves them from cold to warm states).

Similarly, the load-to-meet maps to:

$$\frac{W}{R_W} + \frac{D}{R_D} > \lambda_{max} \tag{2}$$

where R_W is the warm sandbox reuse period, and R_D is the dedup sandbox reuse period.

5.2.2 Platform Efficiency and Latency Measures

Denoting the memory footprint of warm sandboxes as m_W, the memory footprint of dedup sandboxes as m_D and the overhead of dedup starts as m_R, we can express the total memory usage of D dedup and W warm sandboxes as:

$$M = W \times m_W + D \times (m_D + m_R) \tag{3}$$

If all the dedup and warm sandboxes on the platform were used to fulfill N requests in time T, then the average startup latency shall be given by:

$$S = \frac{1}{N}\left(W \times \frac{T}{R_W} \times s_w + D \times \frac{T}{R_D} \times s_d\right) \tag{4}$$

where s_W is the warm startup latency, s_D is the dedup startup latency, and R_W and R_D have the same meaning as above.

5.2.3 Policy Interface

Using the platform constraints mentioned above, as well as efficiency and performance metrics, our framework can provide easy access to the providers to control these metrics while meeting the constraints. For example, in Medes, providers can configure the policy in two ways (combinations of these can also be configured trivially):

Meet an average startup latency target. Suppose the target is $\alpha \cdot\ s_W$, where $\alpha > 1$. In this case, the policy optimally keeps sandboxes so as to occupy least memory footprints while meeting the latency targets:

$$\begin{aligned} &\underset{W,D}{\text{Min}}\ M \\ &s.t.\ S < \alpha\ s_W \\ &\qquad \text{and constraints 1, 2 are satisfied.} \end{aligned}$$

Limit the cluster memory usage. Suppose the maximum desired memory usage is M_0. The policy optimally manages

sandboxes so as to get the best startup latency, using the following optimization problem:

$$\underset{W,D}{\text{Min}}\ S$$
$$s.t.\ M < M_0$$
and constraints 1, 2 are satisfied.

The solution to the above optimization problem acts as a guidepost for the decisions of the sandbox management policy. If the solution is feasible (i.e., the above system of equations is consistent), the policy computes decisions for each function in order to converge to the optimal number of dedup sandboxes for that function. If, however, the solution is infeasible (for example, if the solution gives negative values for D or $D > C$), the policy aggressively employs deduplication and keeps the sandboxes warm only if enough memory is available and the available sandboxes are not enough to suffice the request rate.

5.3 Multi-function Policy

The sandbox management policy described in Section 5.2 gives the policy for a single function. This has the advantage that the provider can regulate memory and performance metrics for each function separately. For example, critical functions can be run on a tight latency constraint while best-effort functions can be run on a loose latency constraint. Additionally, the provider may also want to limit the overall serverless platform memory usage. Medes can also support memory constraint for multi-function workloads by dividing the total memory budget between functions in proportion of their average request arrival rates.

6 Implementation

We build a prototype for Medes in C++ (~6K sloc). We use Docker as the sandbox environment. We implement the two core components of Medes - a global controller and a service module for each machine. The controller consists of three major components - the scheduler, policy module, and the fingerprint registry. The service module consists of three components - the daemon, the dedup agent (to deduplicate memory states and restore them), and the RDMA module (for making remote memory accesses). We use REST APIs to interact with the Docker daemon. The dedup agent and the controller interact with each other using protocol buffers [5]. Additionally, Section 4 discusses the various implementation choices we made related to dedup and restore ops.

7 Evaluation

We evaluate the performance benefits of Medes and its flexibility in navigating the performance-cost trade-off by answering the following questions:

- Can Medes improve the function startup latencies? (Section 7.2)
- Can Medes help reduce the overall memory footprints? (Section 7.3)
- Can tuning fixed keep-alive policies achieve the same performance-memory tradeoffs as Medes? (Section 7.5)

Notation	Function Environment	Average Execution Time	Memory Usage
Vanilla	Empty Environment	150ms	17MB
LinAlg	Linear Algebra	250ms	32MB
ImagePro	Image Processing	1200ms	26.4MB
VideoPro	Video Processing	2000ms	48MB
MapReduce	Map Reduce	500ms	32MB
HTMLServe	HTML Serving Application	400ms	22.3MB
AuthEnc	Authentication / Encryption	400ms	22.3MB
FeatureGen	Feature Generation	1000ms	66MB
RNNModel	RNN Model Serve	1000ms	90MB
ModelTrain	Regression Model Training	3000ms	87.5MB

Table 2. Execution time and memory footprint for various functions in the FunctionBench suite.

- How does Medes perform under memory pressure situations? (Section 7.4)
- What are the overheads of using the dedup state? (Section 7.7)

7.1 Experimental Setup

We evaluate Medes on a 20 node cluster on CloudLab [15]. All nodes have 64GB memory and a 10Gbps NIC. One node out of these acts as the controller. No sandbox runs on the controller. The remaining nodes are all accessible via an RDMA network.

Baselines: We compare Medes against state-of-the-art serverless platforms using two baselines - first, we use the fixed keep-alive policy, which is used by several commercial serverless providers such as AWS Lambda as well as open-source platforms like OpenFaas and OpenWhisk. For our experiments, we take a fixed ten-minute duration as the keep-alive period. We evaluate the baseline over a diverse range of keep-warm periods in Section 7.5 and we found ten minutes to have the best performance on our workloads. Additionally, we also compare against an adaptive keep-alive policy [29] which is adopted by Azure Functions, wherein the keep-alive period is chosen based on the request inter-arrival times.

Workloads: For the request arrival patterns, we use arrival patterns from the Azure Function trace [29]. We found that the per function arrival rates were low; hence we scale up the request rates 5×. We use multiple such one-hour traces for our evaluation.

For the function environments, we use all ten functions from the FunctionBench [20] suite (as tabulated in Table 1). The execution time and the memory usage of each of the functions we used are given in Table 2. We construct a full benchmark trace, assigning each function a one-hour trace chosen from the production traces. We use this workload for evaluating performance and memory benefits of Medes in Sections 7.2, 7.3 and 7.4. We use another smaller representative trace for microbenchmarks (Section 7.5- 7.6) and sensitivity analysis (Section 7.8).

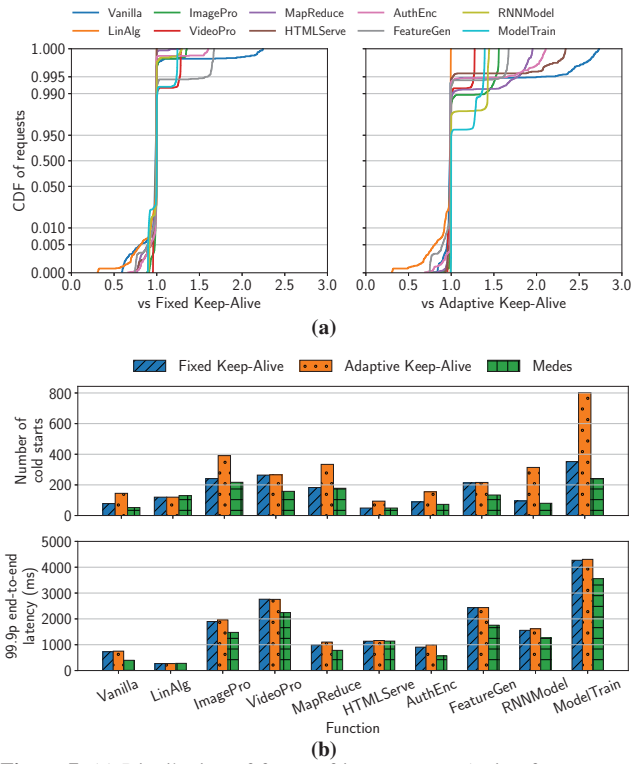

Figure 7. (a) Distribution of factor of improvement (ratio of per-request end-to-end latencies) over Fixed keep-alive and Adaptive keep-alive policies. (b) Function-wise improvements in the number of cold starts and 99.9th percentile of end-to-end latencies.

7.2 Function Startup Time

We begin by evaluating whether Medes can provide better function startup times and, in turn, evaluate the impact on end-to-end latencies.

Methodology: To evaluate the function performance in Medes, we operate the platform policy with latency as the objective function (P1; Section 5).

Additionally, we keep a fixed software-defined limit on the per-node memory usage of the testbed and provide this as the parameter to the sandbox management policy. We use a memory limit of 2GB per node to ensure that the cluster is oversubscribed. This entails that the memory usage of all the different policies remains the same, and we can evaluate the trade-offs by comparing the function performance.

Metrics: Since cold starts impact the tail performance of a system, we use the number of cold starts as a metric for comparison in our experiments. We also evaluate the improvement factor of the end-to-end latencies of Medes compared to those of the baselines on a request by request basis and then show a distribution of this factor of improvement. We evaluate these metrics on a per-application basis.

Figure 7 shows the application-wise performance benefits of using Medes over the baselines. We observe that Medes can provide up to 2.25× and 2.75× improvements in the end-to-end latencies (Figure 7a). These figures also show that for

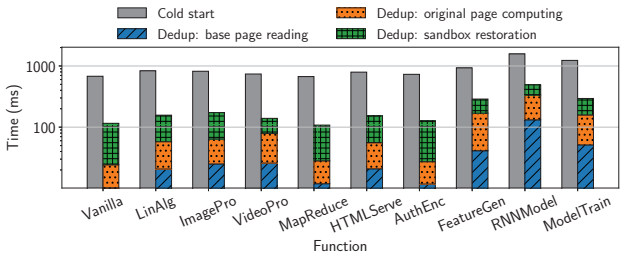

Figure 8. Breakdown of dedup start times vs the cold start times for various applications.

a small number of requests (< 1%), Medes leads to larger end-to-end latencies. We observe that this is because some requests, which would otherwise have been served by warm sandboxes, are served by dedup sandboxes in Medes. However, in the tail Medes provides better performance because the tail performance is impacted by cold starts. Figure 7b demonstrate these improvements in the 99.9th percentile latencies for the ten functions. We observe that Medes gives 1-2.24× improvements in the 99.9th percentile against the Fixed keep-alive policy. Likewise, Medes gives up to 2.3× improvements in the 99.9th percentile against the Adaptive keep-alive policy.

Since the cluster memory pool is chosen to be oversubscribed – this improvement in function startup times demonstrates that Medes is able to strike a better trade-off than the baselines.

7.2.1 Sources of Improvement

The primary source of improvement is the reduction in the number of cold starts for each of the functions (as shown in Fig 7b). We observe that Medes can provide up to 1.85× and 6.2× reductions in the number of cold starts across applications, compared to the fixed and adaptive keep-alive policies, respectively. Because of a significant drop in the number of cold starts, we see a benefit in the tail latencies. This reduction in the number of cold starts is because, on average, Medes deduplicates about 39% of all sandboxes. This deduplication helps Medes to keep 7.74% and 37.7% more sandboxes in memory compared to the fixed keep-alive and adaptive keep-alive policies.

Insights: It is noteworthy that the tail latency improvements not only depend on the number of cold starts but also on the cold start overheads - which varies for different functions. Figure 8 compares the dedup startup latencies against the cold start latencies and shows that the dedup starts are significantly faster than cold starts consistently across all functions. It also breaks down the dedup startup latencies into the three critical phases of sandbox restore - reading base pages, computing original pages, and restoring the dump (as mentioned in Section 4.2). Figure 7b also shows that Medes gains more performance benefits for functions with larger memory usage (e.g., ModelTrain, FeatureGen and RNNModel) - this is because deduplicating more memory-consuming functions gives more memory savings.

Function Environment	Percent Savings
Vanilla	$\frac{4.6MB}{17MB} = 27.06\%$
LinAlg	$\frac{10.5MB}{32MB} = 32.81\%$
ImagePro	$\frac{11.36MB}{26.4MB} = 43.03\%$
VideoPro	$\frac{12.22MB}{48MB} = 25.46\%$
MapReduce	$\frac{5.10MB}{32MB} = 15.94\%$
HTMLServe	$\frac{9.88MB}{22.3MB} = 44.30\%$
AuthEnc	$\frac{4.79MB}{22.3MB} = 21.48\%$
FeatureGen	$\frac{25.67MB}{66MB} = 38.89\%$
RNNModel	$\frac{52.23MB}{90MB} = 58.03\%$
ModelTrain	$\frac{26.33MB}{87.5MB} = 30.09\%$

Table 3. Percent memory savings for each function environment

Enabling ASLR reduces memory savings because a fingerprint size of 5 chunks is insufficient to capture page similarity fully. Specifically, we observe that the average memory savings per sandbox reduced from 28.8MB in the ASLR enabled case to 12.1MB in the ASLR enabled case. However, increasing the fingerprint size (number of chunks in the fingerprint) would get similar memory savings even with ASLR enabled.

7.3 Cluster Memory Usage

In this section, we study the extent to which deduplicated sandboxes save memory and how effective is Medes in identifying the reusable sandbox chunks.

Methodology: To evaluate maximum memory savings possible in Medes, we operate the policy with memory usage as the objective function, and we run the multi-function workload as earlier. Since the fixed and adaptive keep-alive policies do not have any method to ensure that a latency bound is met, we use a tight latency bound for the workload (α in Policy P1 is set to be 2.5).

Total cluster memory usage: Figure 9a shows that Medes uses 11.4% less memory on average compared to the fixed keep-alive policy, while meeting the same latency targets. The adaptive keep-alive policy has a smaller memory usage as its short keep-warm periods lead to reduced memory usage, but that comes at the cost of increased number of cold starts - it incurs at least 50% more cold starts than Medes (see Figure 9b). We further observe that Medes can provide up to 1.58× improvement in the number of cold starts over the fixed keep alive policy, which results in up to 1.9× improvements in the end-to-end tail latencies. This is because the flexible policy implemented by Medes allows it to deduplicate sandboxes of functions with larger memory footprints, making more space to keep warm sandboxes for other (smaller) functions - such that both functions meet their respective latency targets. For example, by aggressively deduplicating sandboxes of RNNModel and not keeping warm ones, Medes reduce the memory usage at the expense of cold starts to the extent that the latency targets are met. The resulting memory savings can also be used to keep more warm sandboxes of other functions. Overall, Medes can meet the latency targets in a smaller memory footprint as compared to fixed keep-alive policies.

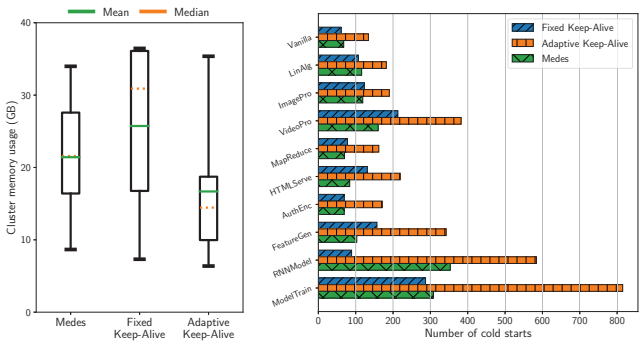

Figure 9. Memory savings on Medes while meeting latency targets.

7.3.1 Sources of Improvement

We attribute the smaller memory footprint of Medes compared to the fixed keep-alive policy to the memory savings due to deduplication. Specifically, we calculate the dedup benefit for each deduplicated sandbox. We calculate the total size of saved bytes (= (page size - patch size)) for all deduplicated pages of a dedup sandbox. Then we report the average duplication of all the dedup sandboxes of a function. We find in our experiments that for the smallest function (Vanilla), our deduplication mechanism leads to a savings of ≈5MB per sandbox, while for the largest function (RNN Model), we can obtain ≈52MB of savings per sandbox. Using this and the average memory usage of a sandbox as mentioned in Table 2, we can calculate how much percent of the sandbox memory actually got removed during the deduplication operation. Table 3 calculates these savings.

Cross Function Duplication: In our evaluation, we observed that among all the pages that were deduplicated - only 32.86 % were deduplicated with a page belonging to the same function, and roughly 67 % were deduplicated with a different function. This cross-function duplication is critical to gain the aforementioned memory savings (Section 2).

7.4 Medes under Memory Pressure

We now study the impact of Medes under memory pressure using the same multi-function workload used in Section 7.2. We decrease the overall memory pool of the platform by decreasing the software limit for the memory available per node. We observe that the benefits provided by Medes relative to the keep-alive baselines increase as the memory pressure increases - in comparison to the fixed keep-alive policy, the number of cold starts are improved by 22% in the no memory pressure case to 37% and 40.67% in the memory pressure cases (see Figure 10). Similarly, the number of cold starts reduces by about 52% in comparison to the adaptive keep-alive policy, in all the three memory pressure situations. This is primarily due to the keep-alive baselines incurring more cold starts relative to Medes as they evict sandboxes under memory pressure whereas with Medes, the memory footprint of sandboxes decreases due to deduplication. We observe that even under extreme memory pressure situations, Medes can keep

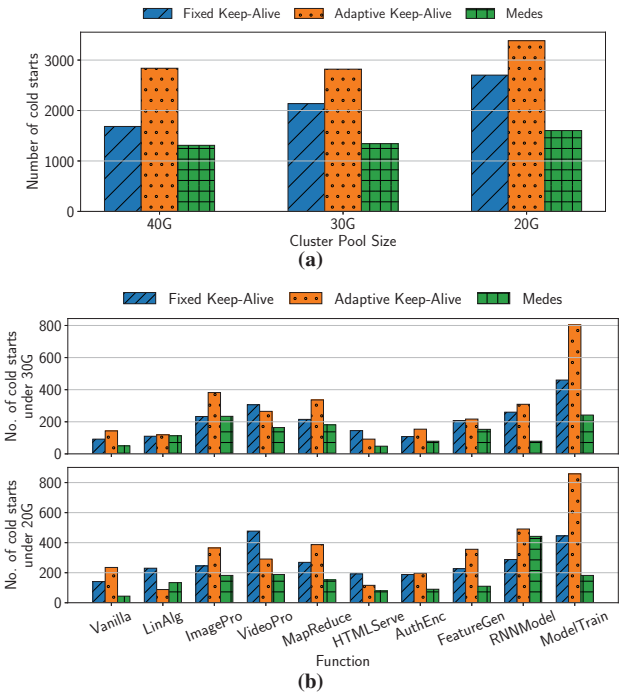

Figure 10. Cold starts incurred by Medes versus the fixed and adaptive keep-alive policies under various scenarios of memory pressure: (a) Different cluster pool sizes, (b) Function-wise breakdown for 30G and 20G cases

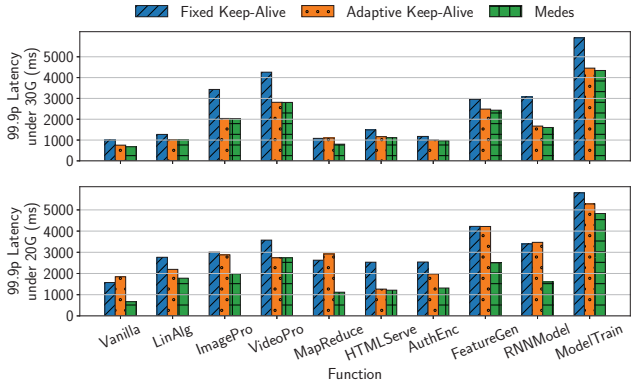

Figure 11. Improvements in end-to-end latencies under (a) small memory pressure, and (b) extreme memory pressure.

42.98% and 55.7% more sandboxes compared to fixed keep-alive and adaptive keep-alive, respectively. This translates into 3.8× improvement in the tail latencies over these baselines under memory pressure situations (see Figure 11). Further, we observe that functions with larger memory footprint and setup overheads see the most benefits due to Medes (see FeatureGen and ModelTrain functions in Figure 11). Deduplicating large sandboxes helps in two ways - first, dedup starts being faster than cold starts lead to improved function slowdowns, and second, it also gives more memory savings - relieving the memory pressure (Section 7.3).

7.5 Medes Vs. Different Keep-Alive Baseline Setting

In Section 7.2, we demonstrate that Medes outperforms a standard fixed keep-alive policy as well as the adaptive keep-alive policy. In this section, we answer the question - 'Can

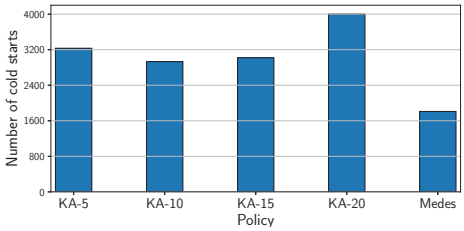

Figure 12. A sweep over various keep-warm periods and comparison with Medes.

varying the keep warm period give the same performance vs. memory trade-offs as Medes?'

To this end, we use a smaller set of the multi-function workload used in Section 7.2, and run several keep-warm policies that make use of various fixed keep-warm periods. We group functions that observed roughly similar benefits (for function startups and memory savings). We choose a representative set of {LinAlg, FeatureGen, and ModelTrain} to carry out this experiment. Figure 12 shows the performance of the various keep warm policies along with Medes. We observe that as the keep-alive period increases from 5 min to 10 min, it leads to a 9.4% reduction in the number of cold starts. However, going from a keep-alive period of 10 min to 15 min and 20 min gives a 3% and 36% increase in the number of cold starts - thereby degrading the performance. We reconcile that beyond a threshold - keeping these sandboxes in memory becomes increasingly prohibitive as a lot of idle sandboxes lead to evictions in the face of memory pressure (for example, keep warm period of 20 minutes leads to more cold starts because sandboxes are evicted before they hit the keep warm timeout). Medes strikes a better trade-off owing to the smaller footprints of deduplicated sandboxes, and gives a 38.2% reduction in the number of cold starts compared to the best fixed keep-alive policy (=10 min).

7.6 Medes + Optimized Checkpoint-Restore

Many recent works have tried to bridge the gaps between warm and cold starts by heavily optimizing the cold startup times using checkpoint-restore mechanisms [8, 13, 26]. All these works target redundancy of function memory state between subsequent invocations. Medes goes one step further and also optimizes the redundancy between functions that are in memory at the same time but in different locations. In this section, we demonstrate that Medes can further improve systems that make use of optimized checkpoint restore mechanisms by reducing the memory footprint needed to keep sandboxes in memory. To this end, we emulate the sandbox template method of Catalyzer [13]. We replace all cold starts in the workload by a sandbox restore. We run this emulated setup with and without Medes on the representative workload used in Section 7.5. Figure 13 shows the improvement in the number of cold starts when memory deduplication is used in conjunction with sandbox restore optimizations. This is due to the heavy deduplication employed by Medes leading to 42.8% of the sandboxes being deduplicated.

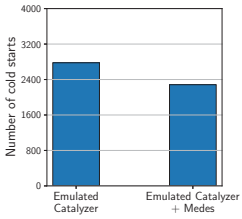

Figure 13. Integrating Medes with Catalyzer.

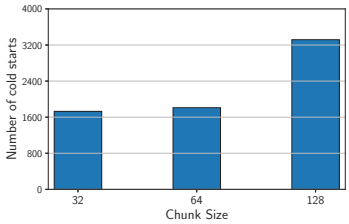

Figure 14. Sensitivity to the Chunk Size Medes.

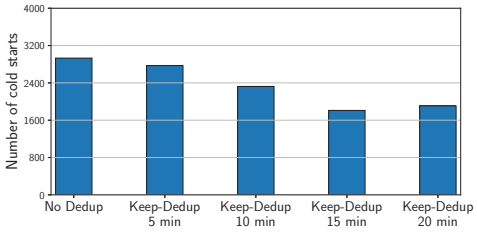

Figure 15. Sensitivity to the keep dedup period.

7.7 Medes Overheads

In this section, we discuss the overheads of Medes at the Dedup Agent and the controller.

Dedup Agent: In our experiments running 5× magnified production traces, we did not observe significant overhead due to Medes. The metadata and base sandbox checkpoint maintained at the Dedup Agent was below 10% of the total memory usage on each node. Further, even including this metadata, Medes can provide a smaller memory footprint compared to the baselines (Section 7.3) - owing to the memory savings of deduplication.

Controller Overheads: While we avoid communication with the controller during the critical path of sandbox restores, these operations still happen in the background. In our experiment running the complete function benchmark, we found that the total time for deduplicating a sandbox varied from 2s for the Vanilla function to 3.3s for the ModelTrain function. This includes sending the page fingerprints over to the controller and performing a lookup on the global fingerprint registry. Specifically, the total time to lookup and identify base pages for all pages of a dedup sandbox took from 130ms for Vanilla (total 4k pages) to 1850ms for ModelTrain (total 22k pages). This amounts to a processing time of ~80μs per page in our single-threaded implementation. To further reduce these overheads, the lookups can be parallelized given they are independent (Section 4.3).

Further, we observe that compared to the baselines, the memory usage at the controller only increases by 11.8%, due to the addition of fingerprint registry and policy metadata.

7.8 Sensitivity Analysis

We now understand the sensitivity of Medes to the following aspects of the design using the representative multi-function workload, as described in Section 7.5.

Chunk Size: In Medes, we use a chunk size of 64 bytes for RSCs. Medes samples 64B chunks by value from each page. Thus, the redundancy identification granularity is at 64 bytes. We evaluate the impact of choosing a larger or smaller chunk size for RSCs. Figure 14 shows the number of cold starts incurred by Medes for chunk sizes of 32B, 64B and 128B. We observe that using a larger size chunk reduces the benefit of deduplication and the average memory savings per sandbox drop to 22.8MB (compared to 28.8MB for 64B). Hence, smaller memory savings lead to more evictions and more cold starts. On the other hand, using a 32B chunk leads to hash collisions on the fingerprint table - causing dis-similar chunks to be labeled similar, which again leads to ineffective deduplication. This is evident by the average patch size, which increases to 940 Bytes (from 611B for 64B chunks).

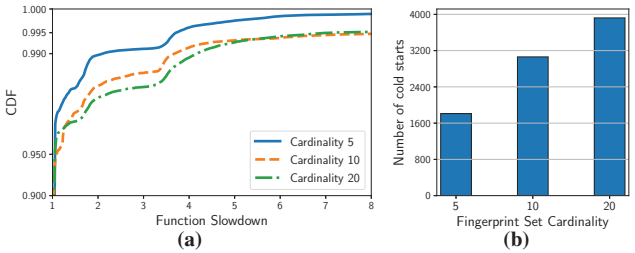

Figure 16. Sensitivity to the fingerprint set cardinality.

Keep-Dedup Period: The keep-dedup parameter determines how long a sandbox remains deduplicated. We vary the keep-dedup parameter from 5 to 20 mins in increments of 5mins. Figure 15 show that initially, as the keep-dedup period increases, the number of cold starts improves by 10%-38% due to the reduction in cold starts as there are dedup sandboxes available. We observe that a longer keep-dedup parameter implies that a dedup sandbox shall be around for a longer time, courtesy of which 38% of the requests that would otherwise incur cold starts now incur faster dedup starts. However, we see that beyond a threshold, the benefits due to Medes decrease (see Figure 15 Keep-Dedup-20mins), as deduped sandboxes are kept around for unnecessarily long times – resulting in more cold starts due to memory pressure.

Fingerprint Set Cardinality: The fingerprint set cardinality corresponds to the number of 64B chunks whose hashes jointly represent the identity of the base page. As expected, with higher cardinality, we get a more accurate representation of the page, leading to higher redundancy identification and elimination, leading to more memory savings — we observed an increase in per-sandbox savings from 28.8 MB to 31.5 MB to 32.54 MB. However, as we increase the set cardinality, we observe that the tail latencies inflate due to more cold starts (see Figures 16a- 16b). This is primarily due to the dedup restore time increasing from 378ms to 478ms to 554ms as the set cardinality increases as more fingerprints lead to more base pages being needed for restoration.

8 Related Work

Serverless Workload Characterization. [21, 24, 31, 34, 38, 39] reverse engineer aspects of serverless platforms by observing the visible metrics. [29] characterizes workloads from Azure Functions. Complimentary to these efforts, we look

at the degree of memory redundancy present in serverless workloads.

Memory Deduplication. Prior works have developed techniques for memory deduplication to reduce memory requirements for virtual machines. They have also been deployed in popular production systems ([36] in VMWare and [7] in the Linux kernel). Such inter-VM deduplication approaches employ coarse-grained intra-machine page sharing. Both [36] and [7] employ memory scans to identify identical pages and remap them. These approaches impose high deduplication overhead as they require frequent scans over large chunks of memory. Further, these approaches are insufficient to realize the full performance benefits of the memory redundancy that exists at the sub-page chunk granularity and the inter-machine memory redundancy. Both of which are exploited by Medes, without making expensive memory scans or any guest OS modifications. [18] is an inter-VM approach that also uses sub-page deduplication similar to Medes. However, the deduplication by Difference Engine falls short as it takes chunks at random offsets in a page to act as the fingerprint. Medes uses value-sampled fingerprints, which is more effective at identifying sub-page level small-sized redundant chunks [9]. Additionally, Medes reboots this sub-page deduplication mechanism to deduplicate sandboxes on-demand on a serverless platform efficiently while dealing with the challenges of the platform's scale, deduplicating and restoring sandboxes on-demand, and performing fast sandbox restores.

Reducing Sandbox Overheads. [29] proposes setting keep-alive windows in a workload-aware manner. Likewise, inspired by traditional caching, [16] proposes a workload-aware keep-alive policy that considers additional function characteristics such as function size and initialization costs. However, they are still required to choose between performance or efficiency and offer little flexibility. In contrast, through the introduction of the dedup state, Medes improves the flexibility of navigating the memory-performance trade-off.

[11, 23] propose using unikernels to reduce overheads. However, this limits their practical adoption due to code compatibility issues [8]. Photon [14] demonstrates memory redundancy between invocations of the same function and proposes to co-locate concurrent invocations of the a function on the same sandbox. Hence, Photon exploits redundancy for functions in execution (while running user code) while sacrificing isolation. A similar approach is used by [19] to optimize inter-function calls by trading off isolation as it proposes running multiple chained functions in the same container. Medes takes an aggressive approach to deduplication and reduces memory footprints for warm functions (even when the sandbox is not executing user code) between sandboxes of not just the same function but different functions as well.

Function snapshotting proposals (Catalyzer [13], REAP [35]) reduce the startup overheads by restoring a sandbox from a snapshot, either stored on disk or shared in memory. Catalyzer employs an on-demand page restore mechanism, where only minimal pages are loaded at cold-start, and subsequent pages are fetched via page faults during execution. While this reduces cold start latencies, it significantly increases function execution times [35]. On the other hand, REAP [35] avoids in-execution page faults by pre-fetching all the pages of a function working set. The basis is again that invocations of the same function have nearly the same working set. Pre-fetching is an expensive task but the assumption is that such pre-fetching is needed infrequently. However, this assumption is limiting for the fast-evolving serverless environment with increasing support for microservices that may require frequent invocations [37]. These works reduce memory overheads for each function type independently and within a machine. In contrast, Medes can reduce memory overheads and hence, improve startup latencies by deduplicating across different function types without any additional function execution costs. In our evaluations, we observed that using Medes in combination with these snapshotting mechanisms can further improve their performance (Section 7.6).

[10, 26] leverage fork-based techniques to reduce overheads. [2, 22] aim to use a sandbox per trust domain. However, these proposals by design still have limited flexibility to navigate the performance-memory trade-off space and sacrifice isolation during request execution.

9 Summary

We propose a new serverless framework, Medes, that breaks the rigid trade-off between memory efficiency and performance in today's platforms. We leverage the fact that warm sandboxes have a high fraction of duplication in their memory footprints and introduce the notion of a redundant sandbox chunk and a new dedup sandbox state. We develop algorithms to identify and eliminate duplication at low computation and memory costs, move sandboxes to dedup states, and quickly restore dedup sandboxes. We then propose a simple sandbox management policy that allows operators to flexibly meet memory efficiency and performance targets. Our results show that Medes results in up to 1×-2.75× lesser end-to-end latencies on real serverless workloads while occupying the same memory footprints as the state-of-the-art alternatives. At the same time, Medes can meet the same latency targets as the fixed keep-alive policy while occupying 11.4% lesser memory on average. These benefits are primarily due to 10-50% fewer cold starts, which we achieve by keeping 7.74-37.7% more sandboxes in memory by heavily deduplicating idle sandboxes.

Acknowledgements

We would like to thank the anonymous reviewers of EuroSys'22 and the members of UTNS Lab for their regular insightful comments and suggestions. This research was supported by NSF Grants CNS-1565277, CNS1719336, CNS-1763810, CNS-1838733, by gifts from Google and VMware, and a Facebook faculty research award.

References

[1] Xdelta3 Compression Library. http://xdelta.org/.

[2] Chromium Projects. https://www.chromium.org/developers/design-documents/site-isolation, 2019.

[3] CRIU. https://criu.org/, 2021.

[4] OpenWhisk. https://openwhisk.apache.org/, 2021.

[5] Protocol Buffers. https://developers.google.com/protocol-buffers/, 2021.

[6] Squeezing the milliseconds: How to make serverless platforms blazing fast! https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e9951bd0, 2021.

[7] How to use the Kernel Samepage Merging feature. https://www.kernel.org/doc/Documentation/vm/ksm.txt, 2022.

[8] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[9] Bhavish Agarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. Endre: An end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pages 419–432. USENIX Association, 2010.

[10] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards high-performance serverless computing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 923–935, 2018.

[11] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[13] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo 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*, pages 467–481, 2020.

[14] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 45–59, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[16] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, pages 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. Swayam: Distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 109–120, New York, NY, USA, 2017. Association for Computing Machinery.

[18] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, December 2008. USENIX Association.

[19] Zhipeng Jia and Emmett Witchel. 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*, ASPLOS 2021, pages 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019.

[21] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 789–794, 2018.

[22] James Larisch, James Mickens, and Eddie Kohler. Alto: lightweight vms using virtualization-aware managed runtimes. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pages 1–7, 2018.

[23] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[24] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410. IEEE, 2017.

[25] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, USA, 2013. USENIX Association.

[26] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[27] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[28] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. Association for Computing Machinery.

[29] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark

Russinovich, and Ricardo 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)*, pages 205–218. USENIX Association, July 2020.

[30] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 138–152, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Arjun Singhvi, Sujata Banerjee, Yotam Harchol, Aditya Akella, Mark Peek, and Pontus Rydin. Granular computing and network intensive applications: Friends or foes? In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 157–163. ACM, 2017.

[32] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. Snf: Serverless network functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 296–310, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 87–95, New York, NY, USA, 2000. Association for Computing Machinery.

[34] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 311–327, 2020.

[35] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. *Benchmarking, Analysis, and Optimization of Serverless Function Snapshots*, pages 559–572. Association for Computing Machinery, New York, NY, USA, 2021.

[36] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, dec 2003.

[37] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.

[38] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 133–145, USA, 2018. USENIX Association.

[39] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 523–536, New York, NY, USA, 2015. Association for Computing Machinery.