Navigating Performance-Efficiency Tradeoffs in Serverless Computing: Deduplication to the Rescue!

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

Navigating the performance and efficiency trade-offs is critical for serverless platforms, where the providers ideally want to give the illusion of warm function startups while maintaining low resource costs. 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 Medes, a serverless framework, that allows operators to navigate the trade-off space smoothly. Our approach takes advantage of the high duplication in warm sandboxes on serverless platforms 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 use innovative techniques to identify redundancy with minimal overhead, and provide a simple management policy to balance performance and memory. Our evaluation demonstrates that Medes can provide up to 3.8× better end-to-end latencies and reduce the number of cold starts by 10-50% against the state-of-the-art baselines.

Keywords: Serverless, Memory Deduplication, Cloud Computing, Virtualization

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 (e.g., user interaction) and launched to execute in a sandbox (e.g., a container) with the needed libraries and dependencies loaded. The platform scales function instances based on invocation rate. Serverless computing has become popular owing to the reduced developer burden in resource management and its pay-per-use model.

Serverless providers must balance performance requirements with resource efficiency as they handle demanding workloads. Fast function instance start times are critical to performance, and efficient resource usage requires matching active sandboxes to demand.

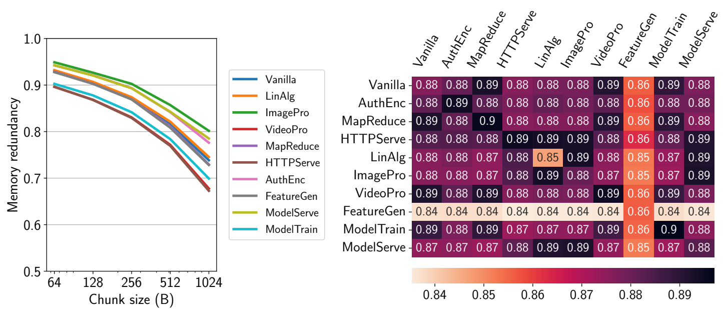
Conventionally, serverless platforms manage performance and efficiency by toggling sandboxes between two states: cold and warm (or paused). Cold sandboxes induce long startup delays (typically in order of seconds [5, 7, 8, 10, 14, 16, 18]), while warm sandboxes kept in memory enable faster reuse but come with resource expense. This approach creates challenging trade-offs between performance and efficiency, making it difficult for operators to control and tune resource use for optimal performance.

In our paper published at EuroSys'22 [15], we present a new mechanism that improves the flexibility of the trade-off space, allowing for better performance and efficiency than the platforms today, while providing operators with a simple way to navigate the trade-off space. Our work improves the trade-off space by introducing a new sandbox state with a memory footprint and startup performance in between those of cold and warm states. The new sandbox state that we 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. The dedup state is built on extending the "reusable sandbox" construct that underlies the warm state today to that of a *reusable sandbox chunk* (RSC).

Reusable Sandbox Chunk: An RSC corresponds to any memory chunk of warm sandboxes that can be "re-used" by other sandboxes. Our empirical study [15] shows that significant duplication exists in the memory states of warm sandboxes: (1) sandboxes of the same function have upto 85% duplication in their memory state; (2) even across sandboxes of different functions, we observe upto 80-90% duplication.

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 (possibly 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 work presents Medes (**Me**mory **De**duplication for **S**erverless), a novel serverless framework that incorporates the dedup sandbox state. However, identifying chunk-level duplication between the memory states of two sandboxes, and exploiting said redundancy on a large serverless platform, spanning across multiple nodes is a challenging task. Each node can have a large number of sandboxes in memory at the same time and each sandbox can have tens of thousands of pages in its memory state. Medes designs an efficient mechanism to perform this deduplication (details in Section 3).



(a) (b)

Figure 1. Memory redundancy in serverless workloads: (a) Between sandboxes of the same function w.r.t. chunk size. (b) Cross function redundancy of functions on vertical axis w.r.t. those on horizontal axis (64B chunks).

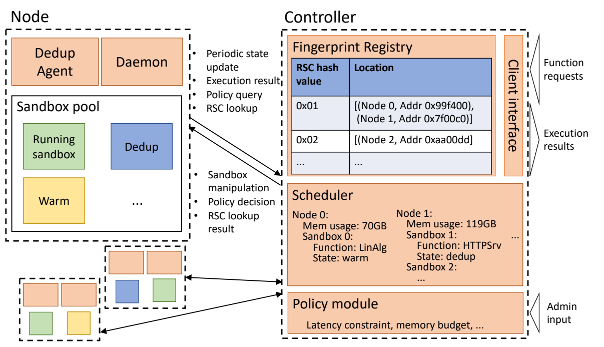


Figure 2. Medes Architecture.

The deduplication mechanism in Medes ensures that the dedup state has a significantly smaller memory footprint than warm sandboxes and that dedup startups are significantly faster than cold starts. Deduplication leads to smaller memory usage and the resulting memory savings can be used to keep more sandboxes, leading to improved performance. Thus, the dedup state can improve the flexibility of the memory-performance trade-off in serverless computing.

Medes exposes a simple interface to jointly control the number of warm *and* dedup sandboxes in memory through which operators can: (a) reason about the performance achieved for a given memory budget, (b) control performance (memory) by directly adjusting memory footprint (performance goals), and (c) customize the policy for different serverless functions.

Our evaluation of Medes against state-of-the-art keep-alive-based baselines on real-world serverless workloads shows that Medes can provide 1×-2.75× smaller tail latencies and 10-50% fewer cold starts compared to baselines. We achieve this by heavily deduplicating warm containers, leading to 7.74-37.7% more sandboxes in memory compared to the alternatives. Crucially, these benefits are enhanced under memory pressure, where Medes provides up to 3.8× improvement in the end-to-end latencies.

2 Medes Overview

2.1 Duplication in Sandbox Memory States

We compare the memory state of sandboxes corresponding to several real-world serverless functions. We use the FunctionBench [13] suite which consists of python serverless functions corresponding to various common use cases. The memory state is obtained by checkpointing sandboxes using CRIU [1].

Same Function Sandboxes. Figure 1a demonstrates that there is significant redundancy — as high as 90% — in sandboxes belonging to the same function. An interesting observation here is 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.*

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 1b 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.

2.2 Medes Architecture

We now desribe the system design for Medes that enables it to exploit the heavy redundancy in memory states. Figure 2 shows the architecture of Medes. Medes consists of a controller and several nodes where functions are executed, interconnected by a cluster/datacenter network.

Medes Controller: The controller has four major components: 1) the interface to clients; 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), and assigns incoming request to an existing or new sandboxes; 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, Medes adds the latter two components to support deduplication, to the controller used by common serverless platforms [2].

Medes Nodes: Each node consists of 1) the daemon that manipulates local sandboxes upon the controller's directives; 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.

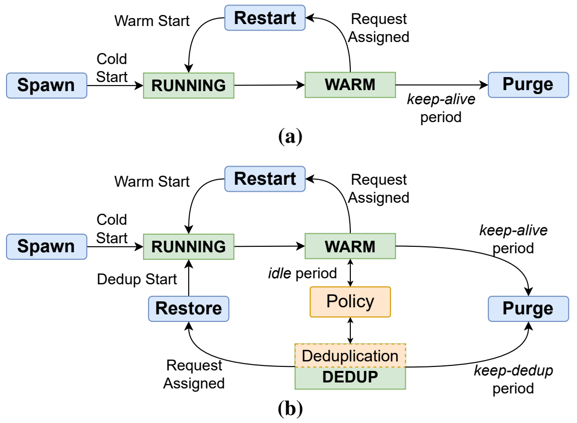


Figure 3. Lifecycle of a sandbox running on (a) Existing Platforms (b) Medes

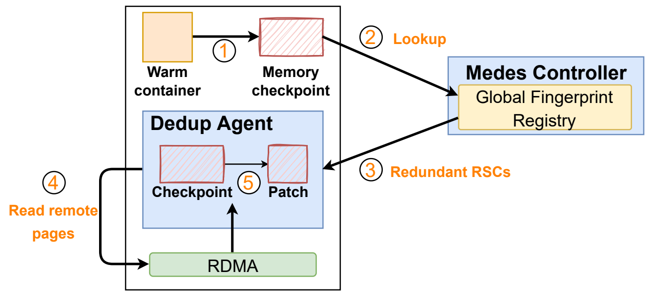


Figure 4. Medes Workflow of the Deduplication Mechanism: 1. Sandbox checkpoint gives the dump of the memory state. 2. RSCs from the memory state are sent to the global hash table on the controller for lookup. 3. The controller sends back the information about redundant RSCs. 4. The Dedup Agent reads 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.

2.3 Sandbox Lifecycle

To offer perspective into Medes' operation, Figures 3a and 3b contrast the sandbox lifecycle on existing platforms and Medes.

The client submits a function request to the controller's interface. The scheduler chooses an available sandbox that can run the function, and hands over the request to the daemon on the chosen sandbox's node. If no sandbox is available, the scheduler spawns a new sandbox. Upon completing execution, the sandbox goes into a warm state, and is removed at the expiry of a 'keep-alive' period or if it is evicted in the face of memory pressure.

Instead of purging the sandbox after a single keep-alive period, Medes allows running a custom policy to determine whether the sandbox is to be transitioned into warm state or dedup state in order to manage memory and performance. This policy module is invoked periodically by the dedup agent. Medes introduces two knobs for this purpose.

The first is called the 'idle period'. Upon expiry of this period, the Dedup daemon checks with the Medes controller whether to dedup the sandbox or keep it warm. If the controller decides to dedup, the dedup agent invokes the *dedup operation* (Section 3.1). During the dedup operation, the agent checks the chunks against the RSC hash values in the fingerprint registry to remove redundant parts of the state, and records RSC locations (obtained from the registry) locally. These 'dedup' sandboxes can later be invoked for incoming requests by performing the *restore operation* (Section 3.2), in which the memory pages are reconstructed by reading the recorded RSC locations.

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.

3 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 multi-tenant requests, whose load can grow arbitrarily.

We briefly explain how these operations are optimized for large-scale systems here. More details can be found in the full paper [15].

3.1 Dedup Op Deep-Dive

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 4). The local dedup agent on the machine initiates a memory checkpoint of the warm sandbox and interacts with the controller to identify duplicate memory chunks (redundancy identification) by sending 'page fingerprints' for each memory page to the controller. The controller uses these fingerprints to find the best 'base page'. The dedup agent then deduplicates each page with its base page (redundancy elimination). Thus, Medes reduces the memory footprint of deduplicated sandboxes by only maintaining patches (including unique leftover pages and memory chunks). We explain 'page fingerprints' and 'base pages' below.

3.1.1 Page Fingerprints.

For each page to be deduplicated, we use a small subset of 64B memory chunks, *value sampled* based on the last two bytes of the chunk, to act as a *fingerprint* for that page. We scan the page over a rolling 64B window and include a 64B chunk in the fingerprint if its last two bytes match a specific pattern. This approach has been used for VM page similarity and redundancy elimination by prior works [4, 12]. Medes renews it for large serverless clusters.

We use five such value-sampled chunks per page. This unordered set then acts as a fingerprint of the page. The number of overlapping fingerprints between two pages represents the similarity between the two pages. Value sampling is computationally lightweight as it involves a single linear scan and an equality check over two bytes. Thus, we eliminate the need to check for each 64B chunk separately, thereby reducing the communication and latency of the Dedup Op.

3.1.2 Base Pages.

For each sampled memory chunk, Medes looks it up in the fingerprint registry to find corresponding RSCs. Each RSC points to a page in memory, giving a set of candidate pages for each memory page. From this set, Medes selects the best candidate page as the "base page" for the respective dedup page. The base page is chosen based on the maximum number of duplicate chunks amongst the sampled chunks, and if there are ties, the page available locally on the same machine is selected.

3.1.3 Local Page Diff.

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.

3.1.4 Low-footprint Fingerprint Registry.

We now discuss which sandboxes to use to populate the registry. Inserting memory chunks from all sandboxes would cause a memory footprint explosion. Even with sampling, nearly 100K chunks per sandbox remain and storing all of them results in high memory usage as platforms can have thousands of sandboxes at once.

Hence, we demarcate specific warm sandboxes on the platform as ‘base sandboxes’ and only the unique memory chunks from these are stored in the registry. This decision is made because the percentage of memory duplication between any two sandboxes of a given pair of functions remains constant.

3.2 Restore Op Deep-Dive

The restore operation involves reconstructing the sandbox using the stored patches and the corresponding (possibly remote) base pages. Figure 5 shows a pictorial representation of the restoration procedure. The key challenge is to ensure that the reconstruction and restoration process is fast.

Medes employs three techniques towards this goal. First, Medes performs the time-consuming [14] steps of restoration prior to deduplicating the sandbox, leaving only memory state restoration during dedup starts. This approach of performing *offline restore* has been explored by other works as well [8]. Second, Medes keeps all information required to complete the restore operation (e.g., patches and the address of the base page) locally with the Dedup Agent at the machine where the deduped sandbox resides. 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 [19].

4 Sandbox Management Policy

With Medes, our objective is to expose an intuitive interface for the providers to specify the performance expectations on a per-function basis and the efficiency expectations at the cluster level. The platform can then leverage the ability to deduplicate sandboxes to navigate the memory-performance tradeoff.

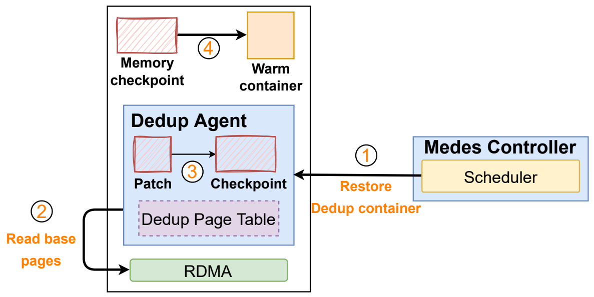


Figure 5. Workflow of the Restoration Mechanism: 1. The scheduler decides when to make a dedup start. 2. The Dedup Agent fetches duplication information about the sandbox from its local data structure. Then, it reads the base pages from the respective nodes. 3. Original (pre-deduplicated) pages of the 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.

4.1 Dedup and Restore Overhead Considerations

Compared to warm sandboxes, dedup sandboxes take additional time to reconstruct the sandbox checkpoint and restore the sandbox memory state from the checkpoint. Furthermore, the reconstruction of the sandbox checkpoing also entails additional memory to read the base pages and compute patches. Hence, frequent dedup starts can lead to memory overheads outweighing the memory savings.

4.2 Optimization Problem

In Medes, we formulate an optimization problem to solve for the number of warm and dedup containers (denoted by W and D respectively). The incoming function load becomes a constraint and Medes can optimize for either the memory usage of the platform, or the function latency, while limiting the other.

4.2.1 Platform Constraints

If the current number of sandboxes on the platform is denoted by C and the desired request arrival rate is denote by λ_{max}, then we have:

$$\mathbb{C}1: \; W + D = C \tag{1}$$

If C is insufficient to handle the load, the controller spins up additional sandboxes. Similarly, the constraint for the function demand can be given as:

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

where R_W is the warm sandbox reuse period, and R_D is the dedup sandbox reuse period. The *sandbox reuse period* is the total time taken for a sandbox to start up and execute a function.

4.2.2 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:

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

If all the dedup and warm sandboxes on the platform were used to fulfill λ_{max} requests in unit time, the average startup latency can be given as:

$$\mathcal{S} = \frac{1}{\lambda_{max}} \left(W \times \frac{1}{R_W} \times s_w + D \times \frac{1}{R_D} \times s_d \right) \quad (4)$$

where s_W and s_D are the warm and dedup startup latencies respectively.

4.2.3 Policy Interface

Using the aforementioned platform constraints, 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:

$$\underset{W,D}{\text{Min}}\ \mathcal{M}\ s.t.\ \mathbb{C}1, \mathbb{C}2, \mathcal{S} < \alpha s_W \quad (5)$$

Limit the cluster memory usage: Suppose the maximum desired memory usage is $\mathcal{M}_0$. The policy optimally manages sandboxes so as to get the best startup latency, using the following optimization problem:

$$\underset{W,D}{\text{Min}}\ \mathcal{S}\ s.t.\ \mathbb{C}1, \mathbb{C}2, \mathcal{M} < \mathcal{M}_0 \quad (6)$$

The solution to the above optimization problem acts as a guidepost for the decisions of the sandbox management policy. Note that the above policy applies *for a single function*. Serverless providers can, therefore, regulate performance and memory usage for each function independently, allowing critical functions to be run on a tight latency constraint, while best-effort functions can be run on a loose latency constraint. Providers can also limit the overall memory usage for multi-function workloads, and Medes can divide the total memory budget proportional to their average request arrival rates.

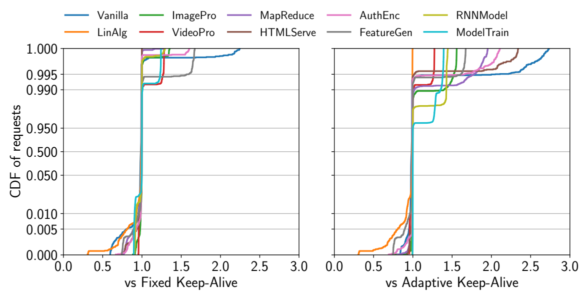
5 Evaluation

The prototype for Medes is developed in C++ and open-sourced at https://github.com/DivyanshuSaxena/Medes. We highlight the performance benefits of Medes and its flexibility in navigating the performance-resource trade-off.

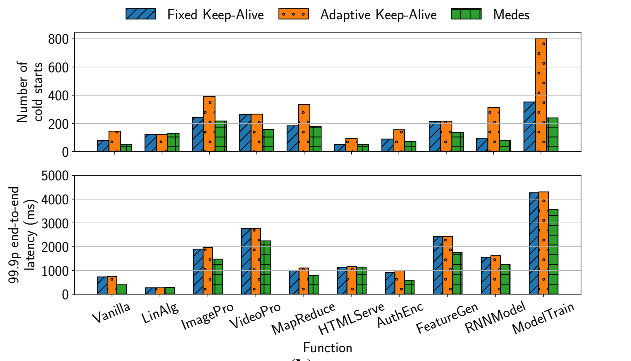
5.1 Experiment Setup

We evaluate Medes on a 20 node cluster on CloudLab [9]. All nodes are accessible via an RDMA network.

Baselines: We compare Medes against state-of-the-art serverless platforms using two baselines. First is a fixed keep-alive policy, commonly used by popular serverless platforms (e.g., AWS Lambda and OpenWhisk). For our experiments, we take a fixed ten-minute duration as the keep-alive period. Second



(a)



(b)

Figure 6. (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.

is an adaptive keep-alive policy [16] (adopted by Azure Functions), wherein the keep-alive period is chosen based on the request inter-arrival times.

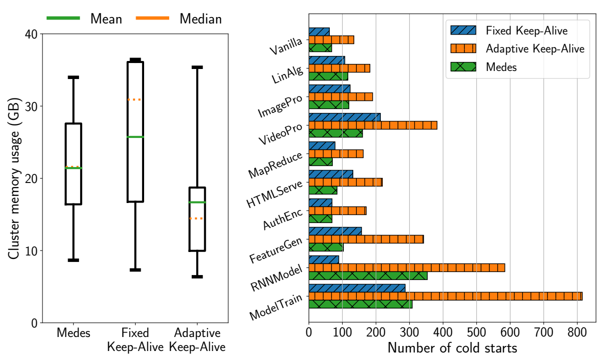
Workloads: For the request arrival patterns, we use multiple one-hour traces from the Azure Function trace [16], scaled up 5×. For the function environments, we use all ten functions from the FunctionBench [13] suite.

5.2 Function Startup Time

Methodology. We operate the platform policy with latency as the objective function (Equation 5). We keep a fixed 2GB software-defined per-node memory limit and provide this as the parameter to the sandbox management policy.

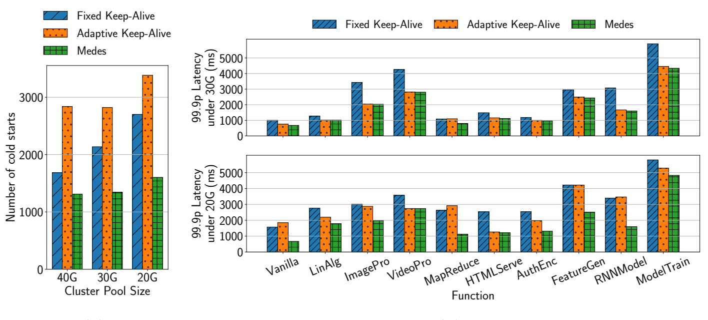
Tail Latencies: We observe that Medes can provide up to 2.25× and 2.75× improvements in the end-to-end latencies (Figure 6a). For a small number of requests ($< 1\%$), Medes leads to larger end-to-end latencies. This is because some requests, which would otherwise have been served by warm sandboxes, 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 6b shows that Medes gives up to 2.24× and 2.3× improvement, for the 99.9th percentile latencies, against fixed and adaptive keep-alive policies, respectively.

Sources of Improvement. The primary source of improvement is the reduction in the number of cold starts. Figure 6b shows that Medes can provide up to 1.85× and 6.2× reductions in the number of cold starts across functions, compared to the fixed and adaptive keep-alive policies, respectively.



(a) **(b)**

Figure 7. Memory savings on Medes while meeting latency targets.



(a) **(b)**

Figure 8. 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

5.3 Cluster Memory Usage

Methodology. We operate the policy with memory usage as the objective function (Equation 6). We intentionally use a tight latency bound for the workload (α in Policy P1 is set to be 2.5). Note that the baselines policies do not have any method to ensure that a latency bound is met.

Total cluster memory usage: Figure 7a 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, but that comes at the cost of increased number of cold starts - it incurs at least 50% more cold starts than Medes (see Figure 7b). Medes' flexible policy allows it to deduplicate sandboxes of large functions, making more space to keep warm sandboxes for other (smaller) functions - such that both functions meet their respective latency targets. For example, Medes aggressively deduplicates RNNModel sandboxes to save memory, sometimes at the expense of cold starts, as long as latency targets are met. These memory savings can then be used to keep other function sandboxes warm.

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. We find in our experiments that for the smallest function (Vanilla), our deduplication mechanism leads to a savings of 5MB ($\approx$27.06%) per sandbox, while for the largest function (RNN Model), we can obtain 52MB ($\approx$58.03%) of savings per sandbox.

Cross Function Duplication. We also observed that of all deduplicated pages, only 32.86 % were deduplicated with a page belonging to the same function, and roughly 67 % were deduplicated with a page from a different function. This cross-function duplication is critical to gain the aforementioned memory savings (Section 2.1).

5.4 Medes under Memory Pressure

We study the impact of Medes under memory pressure. We decrease the overall memory pool of the platform by decreasing the software limit for the memory available per node. Figure 8a shows that in comparison to the fixed keep-alive policy, cold starts are improved by 22%, 37% and 40.67% in the three memory pressure cases. Similarly, in comparison to the adaptive policy, cold starts reduce by about 52% in all three memory pressure situations.

Sources of Improvement. 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. This translates into 3.8× improvement in the tail latencies over these baselines (see Figure 8b).

5.5 Medes Overheads

In our experiments running 5× magnified production traces, we did not observe significant overhead due to Medes at the Dedup Agent. At the controller, the memory usage at the controller only increases by 11.8%, compared to baselines, due to the addition of fingerprint registry.

6 Discussion

Prior works have explored using memory deduplication to reduce memory footprints for VMs on the same node by employing page-level [3, 17] and sub-page level deduplication [12]. These approaches either involve heavy CPU overheads, guest OS modifications or fall short in exploiting full redundancy across nodes of a cluster. Medes presents a clever system design that reboots this sub-page deduplication mechanism to deduplicate 'warm' sandboxes on a serverless platform efficiently while dealing with the challenges of the platform's scale, restoring sandboxes on-demand, and performing fast sandbox restores.

Medes combines page fingerprints with value-sampled redundancy elimination [4] to design an efficient yet effective deduplication method. The hierarchical design of Dedup Agent and Medes controller, and the notion of base sandboxes, alleviate the scalability challenges of exploiting memory redundancy across nodes on a serverless platform.

Can this deduplication approach be applied to other domains? Medes shows that using sufficiently fine-grained chunks (Section 2.1) can lead to heavy duplication [12]. Medes uses this insight to develop a scalable system design that can efficiently utilize this duplication to give memory savings. The resulting memory savings can be used in similar ways to navigate the performance-resource trade-offs, prevalent in other systems as well:

Machine Learning Pre-processing: Cachew [11] shows that storing input data features can be a bottleneck in training pipelines. This is anticipated to become the case for inference in large models as well, for example, language models and GNNs. Along with caching, deduplication can also prove to be a useful tool to optimize systems.

Microservices: Microservice application typically rely on containers to run the application logic. Several service proxies [6] are also deployed along with microservice containers to deal with communication. These proxies are akin to 'warm' sandboxes for Medes, and a similar deduplication approach can reduce overheads there.

A key insight in Medes is that deduplicating warm sandboxes is useful because warm sandboxes are not being used actively. We expect similar characteristics can be identified in ML and Microservice workloads as well - e.g., idle models, less frequently accessed features in ML workloads and idle microservices, can be deduplicated in a similar fashion.

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

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

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

[4] 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.

[5] 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.

[6] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. Leveraging service meshes as a new network layer. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, pages 229–236, New York, NY, USA, 2021. Association for Computing Machinery.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, Carlsbad, CA, July 2022. USENIX Association.

[12] 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.

[13] 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.

[14] 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.

[15] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 714–729, New York, NY, USA, 2022. Association for Computing Machinery.

[16] 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.

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

[18] 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.

[19] 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.