


A motive for memory deduplication in serverless platforms

Thesis in partial fulfillment of MS degree
18th of December, 2020

Mohammed Danish Shaikh



(Aditya Akella)



(Michael Swift)

*To my parents,
Abdul Hameed Shaikh and Kausar Jahan Shaikh*

ACKNOWLEDGMENTS

This thesis would not have been possible without the support and encouragement of a lot of people, to whom I want to thank.

First and foremost, I would like to thank my advisor Aditya Akella for his support and guidance throughout the project. His feedback on my research work was always extremely insightful and it helped shape and develop my research ideas.

I would like to thank Michael Swift for always being available for discussing the blockers of the project and suggesting insightful ideas to get through them. Thanks to Divyanshu Saxena for being a really dedicated and motivated co-worker. It was a pleasure working with him over components of this work.

I am grateful to all my peers and collaborators who contributed their time and insights: Junaid Khalid, Yanfang Le and Arjun Singhvi. In addition, I would like to thank my parents and sisters for their wise counsel and sympathetic ear.

CONTENTS

Contents iii

List of Tables v

List of Figures vi

Abstract vii

1 Introduction 1

2 Background 4

2.1 *Serverless Computing* 4

2.1.1 End User Perspective 4

2.1.2 Cloud Infrastructure Provider Perspective 4

2.1.3 Characterizing Real World Serverless Apps 8

2.1.3.1 Benchmarking Methodology 8

2.1.3.2 Results 9

2.2 *Remote Direct Memory Access* 11

3 Motivation 13

3.1 *Basis* 13

3.2 *Experiments* 14

3.2.1 Design 14

3.2.2 Methodology 15

3.2.3 Setup 16

3.2.4 Results 17

3.2.5 Future Work 20

4 Design 22

4.1 *Architecture* 22

4.1.1	Load Balancer, Database and Scheduler	22
4.1.2	Deduplication Service	22
4.1.2.1	Dedup Controller	22
4.1.2.2	Dedup Client	23
4.1.3	RDMA module	23
4.2	<i>Workflow</i>	24
4.2.1	Deduplicating a container	24
4.2.2	Starting a deduplicated container	24
4.2.3	Evicting a container	25
5	Related Work and Evaluation Methodology	26
5.1	<i>Related Work</i>	26
5.2	<i>Evaluation Methodology</i>	27
6	Conclusion and Future Work	29
6.1	<i>Conclusion</i>	29
6.2	<i>Future Work</i>	29
A	Appendix	31
A.1	<i>Probability of duplication</i>	31
	Bibliography	32

LIST OF TABLES

3.1	Experimental Parameters	16
-----	-----------------------------------	----

LIST OF FIGURES

2.1	Application breakdown	5
2.2	DAG	5
2.3	Serverless Architecture	6
2.4	Serverless Workflow	7
2.5	Distribution of (a) execution time, (b) code size, (c) SNE	8
2.6	Distribution of (a) memory provisioned, (b) execution time, (c) SNE across background and foreground functions	9
2.7	(a) Traditional Interconnect vs (b) RDMA Interconnect	11
3.1	Experiment Architecture	14
3.2	Duplication observed in C_1 for various window sizes with (a) ASLR enabled, (b) ASLR disabled	16
3.3	Duplication observed in C_2 for various window sizes with (a) ASLR enabled, (b) ASLR disabled	18
3.4	Page level (4096 bytes) duplication observed for different configurations of experimental parameters with (a) ASLR enabled, (b) ASLR disabled . . .	19
4.1	Serverless architecture for performing deduplication	23

ABSTRACT

The past few years have seen a widespread adoption of Function-as-a-Service (FaaS) or serverless computing platforms owing to (i) its pay per use model (ii) automatic scalability offerings (iii) increased use of micro-services to build web applications. While FaaS eases the deployment and scalability of applications for application developers, it also introduces new challenges in resource management for the cloud provider. The serverless systems of today suffer from the issue of “cold start” which causes unpredictable and increase in the execution time of serverless functions. Such an increase can be a factor of magnitude higher as compared to the serverless function execution time in the non cold start scenarios. Hence, there have been a wide line of research trying to reduce cold start by either: (i) developing better scheduling algorithms for serverless workloads (ii) improving the start up time of the runtime used to execute serverless functions (iii) packing the memory more efficiently.

Recently, we have also noticed a surge in the usage of RDMA in systems research, specifically leveraging it’s one sided nature to relief CPU load on big data systems during data transfer. Futhermore, since RDMA offers mechanisms to access remote memory at latencies which are comparable to local memory access latencies, researchers have been looking at making systems more efficient by visualizing distributed system memory as a global memory pool that can be accessed via RDMA at low latencies.

Cloud providers currently do not look at the global memory state of a FaaS system in a content aware manner. In this work, we explore the opportunity for visualizing the memory available in the serverless platform as one combined memory pool and store in-memory images of serverless functions remotely in the face of memory pressure. These images can subsequently be fetched using RDMA. Doing so also enables us to deduplicate pages across images belonging to the same user.

1 INTRODUCTION

In the recent years, microservices architecture has risen in popularity for application development since its modular characteristics lead to flexibility, scalability and reduced development effort [18]. In such an architecture, the application is decomposed into discrete services that implement specific business functions. This leads to “loosely coupled” services which can be built, deployed and scaled independently.

The past few years has also witnessed a widespread increase in data storage and computing requirements owing to an increase in data and users. This has made big data and distributed computing the norm. Subsequently, the time and effort spent on deploying and maintaining such big data applications started becoming significant as compared to that spent in developing the application. This led to the diversion of time and investment of application developers and business organization respectively from focusing on the business logic.

Serverless was invented by cloud providers [5; 20; 9; 40] to enable faster deployment of microservices by letting application developers solely focus on the business logic and handle the scalability and maintenance of the application themselves [19; 51; 26]. Furthermore, the pay per use model offered by serverless is also appealing to the business organizations since they would only pay for the compute and memory resources used for execution of the application.

Cloud providers share the serverless infrastructure across users of the serverless platform. To isolate the utilization of compute and memory resources in such shared infrastructure, cloud providers execute serverless functions in sandboxes (for instance, virtual machines, containers, etc) [1]. These sandboxes also need to be setup when a request for a corresponding serverless function is received for the first time, which adds to the execution time of the serverless function. This is known as cold start. Subsequently,

the cloud providers keep the sandbox around for a finite amount of time so that consequent invocations of the corresponding serverless function does not have to pay the start up cost. This is known as warm start.

Cloud providers have to purge the sandboxes corresponding to unused serverless functions to ensure the efficient utilization of their compute and memory resources, leading to cold starts [16; 43]. Researchers have demonstrated that execution time of such cold started serverless functions can be significantly higher than that of warm started serverless functions [10]. Subsequently, researchers have tried to reduce the cold start by (i) better scheduling for serverless workloads (ii) improving the serverless function runtime.

The amount of time a sandbox corresponding to a serverless function is kept around in the serverless infrastructure leads to a tradeoff between the utilization of memory in the serverless infrastructure and the execution time noticed by the end user. Scaling of serverless functions is handled transparently by cloud providers by instantiating multiple sandboxes for the given serverless function. These sandboxes can be located in the same or different machines in the serverless infrastructure. Such sandboxes are expected to have the exact same memory images, which we hereafter refer to as duplication of type *D1*. Furthermore, over the past two decades, researchers have found duplication between heterogeneous virtual machine runtime images [21]. This implies that there exists some amount of duplication in the runtime images of heterogeneous sandboxes. We refer to this as duplication of type *D2*.

In today's world, when a sandbox is kept around in memory to avoid cold start of a serverless function, other machines in the cluster are not consulted to see the state of the memory of the global system at that point in time. Thus, duplicate contents are stored due to *D1* and *D2* duplication, thereby leading to wastage of crucial memory resources.

Remote Direct Memory Access (RDMA) has been successful in help-

ing scale various applications including key-value stores [17; 28; 29; 34], graph processing systems [44; 54], deep learning [56], and distributed file systems [13; 22; 25; 55]. Due to the unique features offered by RDMA, we believe that the problems imposed by emerging applications on serverless systems can be alleviated by deeply integrating RDMA into the design of serverless systems.

In this work, we aim to study the visualization of the serverless infrastructure as one unified pool of distributed memory. Subsequently, when the time comes to purge a sandbox corresponding to a serverless function, the infrastructure consults with the memory chunks already stored in the distributed memory pool and only stores unique memory chunks. Later, when the in memory state of a deduplicated sandbox needs to be reconstructed for execution of a function, the memory chunks can be accumulated locally by remotely accessing them using RDMA. By doing this, we (i) aim to decrease the overall memory utilization of the serverless platform by eliminating redundancy among the in memory contents stored across the fleet of workers belonging to the serverless platform and (ii) reduce the number of cold starts, and hence the average execution times of the serverless functions.

We start by discussing relevant background of serverless computing and RDMA (Chapter 2) followed by studying the redundancy in the memory contents of sandboxes today (Chapter 3). In Chapter 4, we show the changes that would be required in current serverless systems to perform memory deduplication. Subsequently, we present related work and some pointers on what metrics the aforementioned system should be evaluated on (Chapter 5). Finally, we conclude and mention future directions (Chapter 6).

2 BACKGROUND

This section starts by providing a primer on serverless computing, both from an end user and cloud infrastructure provider perspectives. We then characterize the properties of real world serverless applications available on the repository maintained by AWS [8]. Finally, we provide some background information on RDMA. Note that we use sandbox and container interchangeably hereonwards.

2.1 Serverless Computing

2.1.1 End User Perspective

In serverless computing, the programmer develops an application as a directed acyclic graph (DAG) of functions, uploads it to the serverless platform (which stores the code in a datastore) and registers for an event (e.g., incoming HTTP requests, object uploads) to trigger its execution.

Figure 2.1 shows an application consisting of three microservices. Each microservice is broken down into multiple serverless functions as depicted in the figure. Figure 2.2 further depicts a possible DAG representation of the serverless functions belonging to the application. Note that the specific DAG representation will depend on the interactions between serverless functions modeled by the application developer. These would correspond to the interactions between the microservices in the microservice based design.

2.1.2 Cloud Infrastructure Provider Perspective

At the cloud provider end, the serverless platform consists of a load balancing layer, a scheduling layer, a database layer for storing the serverless functions and a cluster of worker machines (Figure 2.3). When a request

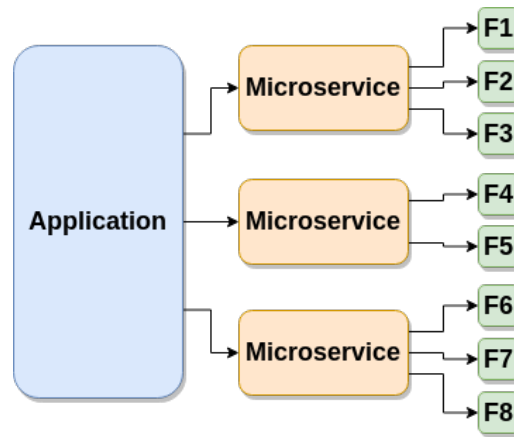


Figure 2.1: Application breakdown

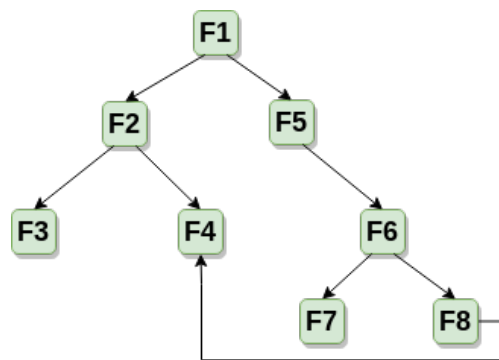


Figure 2.2: DAG

arrives at one of the load balancers, it routes the request to one of the many internal schedulers. The scheduler chooses a worker from the pool of worker machines to execute the function. If the function is being invoked for the first time, then the worker executes the function in following steps (Figure 2.4):

1. Download the code for the function from the database.
2. Start a container environment for executing the function.
3. Download the dependencies into the container.

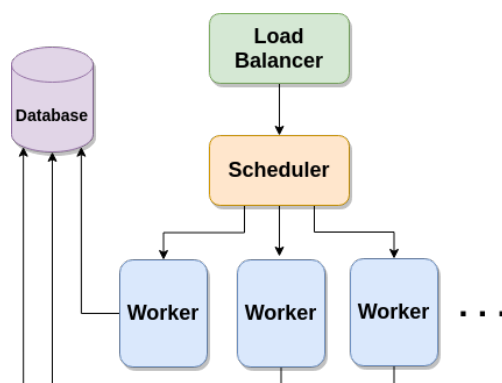


Figure 2.3: Serverless Architecture

4. Execute the function and return the results.

Since serverless functions are inherently designed to execute quickly, the bootstrapping of the container environment for execution of the serverless function adds unnecessary overhead to the execution time of the function. When the cloud provider has to bootstrap the container environment in response to the invocation of a serverless function, this phenomenon is known as a *cold start*.

Given that the cold start overheads are typically unbearable by majority of serverless functions, the cloud providers choose to keep the container environment around in the worker machine which executed the function. This is done in anticipation of additional invocations of the same functions, which can be served by the containers present in the warm state. When a serverless function invocation is served by a container present in a warm state, this phenomenon is known as a *warm start*. In this report, we refer to the threshold amount of time for which the cloud provider keeps a container in the warm state as *keep warm time*.

For a user, if a serverless function always undergoes a warm start, the execution time of the function is uniform. However, the cloud provider cannot keep the containers around indefinitely, since this would cause

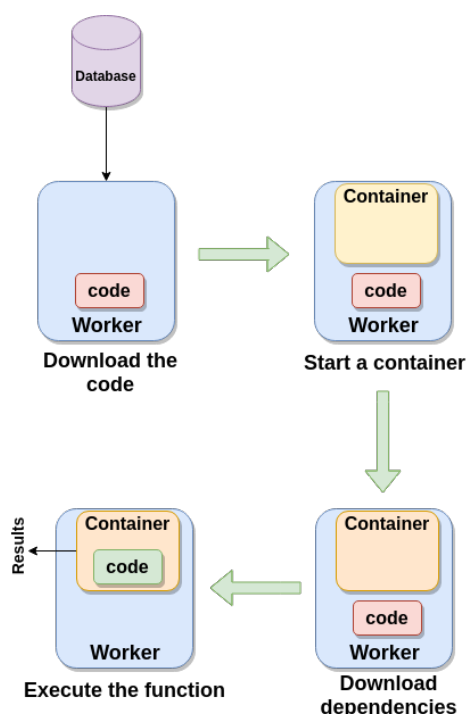


Figure 2.4: Serverless Workflow

inefficiency in the utilization of the cloud provider infrastructure. Yet, at the same time, the cloud providers want to ensure that they provide good service to the end users. The cloud providers choose a middleground nowadays, wherein the container environments corresponding to serverless functions that go dormant after their invocation are kept around for a finite amount of time (for example, AWS Lambda keeps the container environments around for 15 minutes [43]).

In addition to the deployment, the cloud provider is also responsible for handling the scalability of a serverless function, and hence an application consisting of a DAG of serverless functions. Let's say that at a time instant t , n concurrent invocations happen for a serverless function F . Further, let's assume that k warm containers for F already exist in the worker pool of the serverless infrastructure. Then, the cloud provider sets up $n - k$ additional

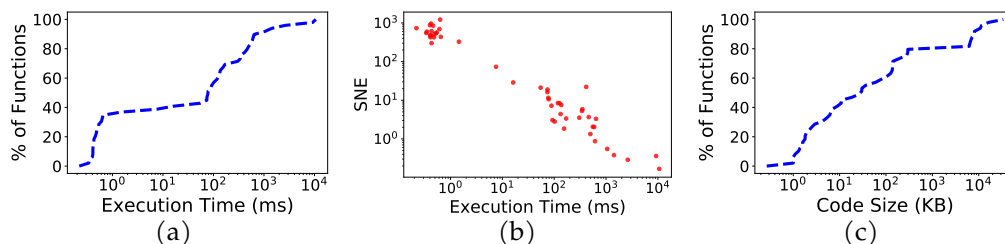


Figure 2.5: Distribution of (a) execution time, (b) code size, (c) SNE

container environments for F for handling the n concurrent invocations at time t .

2.1.3 Characterizing Real World Serverless Apps

We characterize serverless workloads by studying the top 50 deployed apps in the AWS Serverless Application Repository (SAR) [8]. SAR consists of diverse apps that run on AWS Lambda [5]. Internally, AWS Lambda uses Firecracker microVMs [2] to run the apps. These apps typically interact with other AWS services (for example, Redis [6], S3 [7]) as well as third-party services (for example, Slack [46]). This repository is widely used by the serverless community which is evident from the fact that the top app has been deployed 93K times. All 50 apps have a single function, 23 are in NodeJS, 26 in Python, and 1 in Java.

2.1.3.1 Benchmarking Methodology

We use the AWS CLI to upload and trigger the execution of the functions under study. The functions were triggered to run in the us-east-1 region via a VM running in the same region. We collect the following statistics:

1. The *code size* of the function

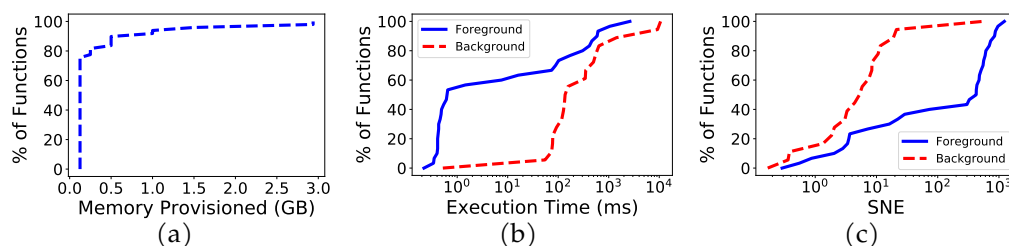


Figure 2.6: Distribution of (a) memory provisioned, (b) execution time, (c) SNE across background and foreground functions

2. *Provisioned memory*, memory available to the function during execution as configured by the programmer while uploading the function to the platform.
3. *Runtime memory*, actual memory consumed during execution of the function.
4. *MicroVM setup overhead*, time taken to setup the function microVM which include the steps discussed above.
5. *Execution time*, time taken to execute the core logic of the function (without including the microVM setup overhead).

2.1.3.2 Results

Mentioned below are the key takeaways from our analysis:

[T1] Majority of foreground functions have execution times less than 100ms. As seen in Figure 2.5a, 57% of functions have an execution time of less than 100ms. These typically correspond to user-facing functions. Figure 2.6b shows the split of execution times based on whether they the functions are foreground or background. As expected, we see that majority (65%) of foreground functions have execution times less than 100ms whereas background functions typically run longer with fewer than 5% having execution times less than 100ms.

[T2] Functions have a wide range of code sizes. As discussed earlier, executing a function also involves downloading the code from the database and setting up the runtime. Prior works have shown that these steps can take a significant amount of time (upto 10s of seconds) depending on the code [37]. In our analysis (Figure 2.5c), we notice that code sizes can be as large as 34MB.

[T3] Sandbox setup overheads dominate execution times. We measure the ratio of the container setup overhead to the execution time of the apps to investigate the impact of overheads on the end-to-end latencies. We refer to this as SNE (sandbox setup overhead normalized by execution time). Figure 2.5b shows that container setup overheads dominate for > 88% of the functions with overhead being > 100X in 37% of them. Our observations are consistent with those of prior work [52; 37; 36]. Figure 2.6c shows that high container setup overheads impact foreground functions much more severely.

[T4] Functions typically have small memory footprints. Figure 2.6a shows the maximum memory provisioned by the functions. We notice that 78% of them require only 128MB.

Based on the aforementioned takeaways, the requirements of an ideal serverless platform is as follows:

[R1] Minimize the impact of containers setup overheads on end-to-end request latencies: Given that these overheads dominate execution times (T3), we wish to eliminate them from end-to-end request execution critical paths. This can be done by (i) improving the start up time of the container environment. (ii) ensuring that requests are served by warm containers as much as possible.

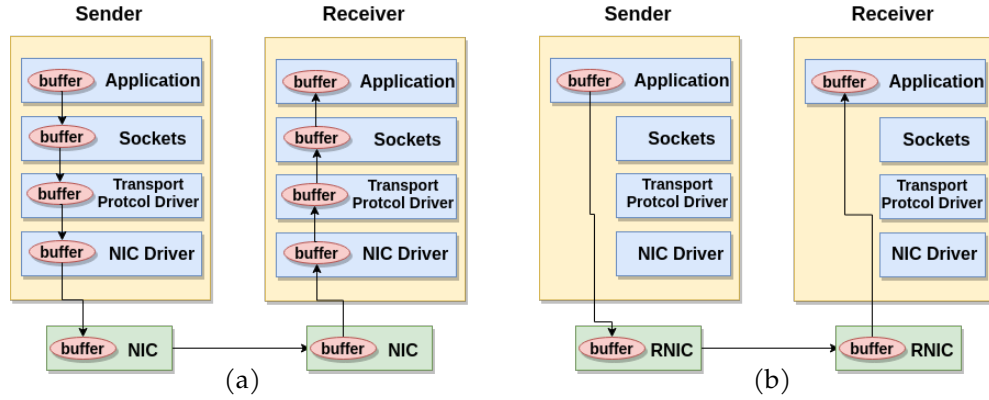


Figure 2.7: (a) Traditional Interconnect vs (b) RDMA Interconnect

[R2] Minimize the impact of control plane overhead on end-to-end request latencies: Given that functions with low execution times are the common case (T1), the load balancing and scheduling layers of the platform are required to take decisions in sub-millisecond scale.

[R3] Have a scalable control plane: Given that many applications will use the platform and their request load can grow arbitrarily, it is required that the load balancing and scheduling layers be scalable.

2.2 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) provides applications with low latency access over a network. The low latency comes from the fact that RDMA is a zero-copy data transfer mechanism wherein the data transfer happens by bypassing the kernel (Figure 2.7). RDMA allows a node to perform one-sided read/write operations from/to memory on a remote node in addition to two-sided send/rcv operations. Both user and kernel level applications can directly issue remote DMA requests (called *verbs*)

on pre-registered memory regions (MRs). One-sided requests bypass CPU on the remote host, while two-sided requests require the CPU to handle them.

Software initiates RDMA requests by posting work queue entries (WQE) onto a pair of send/recv queues (a queue pair or "QP"), and polling for their completion from the completion queue (CQ). On completing a request, the RDMA NIC (RNIC) signals completion by posting a completion queue entry (CQE).

A send/recv operation requires both the sender and receiver to post requests to their respective send and receive queues that include the source and destination buffer addresses.

3 MOTIVATION

3.1 Basis

As discussed in Chapter 2, one of the benefits of a serverless platform is that the cloud provider handles the scalability and deployment of a serverless function. Let's consider two serverless functions F_1 and F_2 , whose container environments have a memory utilization of M_1 and M_2 respectively. Then, it is intuitive that two separate container instances of the same function have almost identical in memory contents. Furthermore, previous research has also observed that there is duplication between in memory contents of different container environments [21].

Let's assume that at a time instant t_1 , n invocations happens for F_1 and m invocations happen for F_2 . In response, the cloud provider would ensure that there exist n and m container environments for executing F_1 and F_2 in it's worker pool of machines. Once the results for these invocations are returned to the user, these container environments are kept around in the worker pool of machines in the warm state, anticipating future invocations. Warm state of a container essentially means that they're *frozen*, and hence don't occupy any CPU resources, but continue occupying memory which contains the in memory state of the container when it was frozen. Let's say that all results are returned to the user at time instant t_2 . So, at t_2 , there exist n containers for F_1 and m containers for F_2 in the warm state. Then, looking at the global memory of the worker machines of the serverless platform, the total duplication in the memory contents across the worker machines at time instant t_2 can be given by the equation:

$$D = (n - 1) * M_1 + (m - 1) * M_2 + \alpha_1 * M_1 * n + \alpha_2 * M_2 * m \quad (3.1)$$

In equation 3.1, α_1 represents the proportion of memory contents of the container environment for F_1 that is duplicated with the memory contents

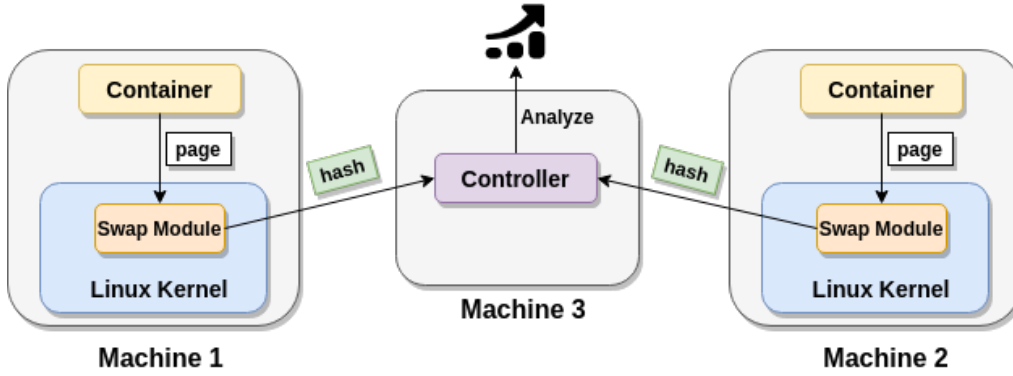


Figure 3.1: Experiment Architecture

of the container environment for $F2$, whereas α_2 represents the proportion of memory contents of the container environment for $F2$ that is duplicated with the memory contents of the container environment for $F1$.

We observe that the cloud providers today do not handle such duplication since they do not look at the global memory pool of the serverless infrastructure in a content aware manner.

3.2 Experiments

We wanted to understand the actual duplication between different instances of same and different container environments. Hence we performed experiments using *LXC* containers [32] in Linux.

3.2.1 Design

The architecture of the experiment is as depicted in Figure 3.1. We hack into the swap module of the Linux kernel to calculate the *md5* hashes of all sliding windows of size S of the page to be swapped. Subsequently, we send hash values ending with *zero* to a remote *controller* for analysis. The memory pages of a container are forced to swap out, the mechanism to

do so is described in Section 3.2.2. The controller analyzes the hash values sent from different machines to finally come up with the duplication number. Since md5 hash values are randomly and uniformly distributed, performing analysis over hash values that end with zero provides a probabilistically good coverage [49]. Similarly, we also perform page level experiments wherein we send the calculate and send the hashes of entire pages for the controller to perform analysis. Since the number of pages will be small, we do not need to cut down on the number of hashes we send over for analysis as we do with sliding window based experiments.

3.2.2 Methodology

The experimental steps undertaken at each machine are as follows:

1. Start a LXC container.
2. Dynamically increase it's memory allocation to a high value (let's say 100MB).
3. Download dependencies to execute a corresponding python function (let's say pandas).
4. Import the aforementioned dependencies and run a python function in an infinite loop. This is done to ensure that the dependencies are loaded in memory when the container in memory state is forced to swap out.
5. Freeze the container.
6. Dynamically change the memory allocation of the container to 0MB. This step triggers a swap out of the in memory pages belonging to the container.

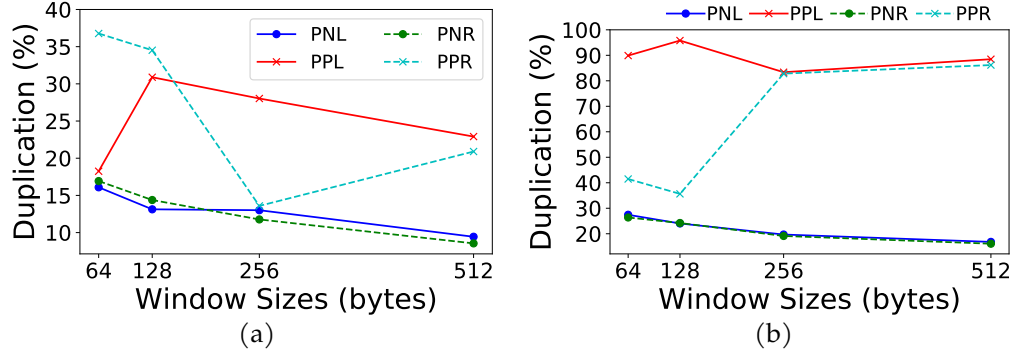


Figure 3.2: Duplication observed in C_1 for various window sizes with (a) ASLR enabled, (b) ASLR disabled

3.2.3 Setup

The experiments were performed on machines in cloudlab [14]. Based on the experimental design and methodology stated above, the experimental parameters as mentioned in Table 3.1 were considered. We perform

Parameter	Values
Container (C)	C_1 and C_2
Function dependencies in Container C_i (F_i)	Pandas (P), Numpy (N)
Sliding window size (S)	64, 128, 256, 512 bytes
Machine placement (P)	Local (L), Remote (R)

Table 3.1: Experimental Parameters

experiments for all combinations of parameters mentioned in Table 3.1. Furthermore, we also noticed that the duplication observed depends on whether Address Space Layout Randomization (ASLR) [4] is enabled or disabled. Hence, we perform the experiments and present the results in both configurations.

3.2.4 Results

Based on the parameters mentioned in Table 3.1, we use the following notations in our graphs while presenting the results:

1. *Figure 3.2:* For each configuration of ASLR, this figure depicts the fraction of duplication observed in the memory contents of C_1 for different experimental configuration over various window sizes. An experimental configuration is mentioned as F_1F_2P . For example, PNL signifies that the dependencies considered for functions in C_1 and C_2 were Pandas and Numpy respectively, and that the containers were spawned in the same machine (local placement).
2. *Figure 3.3:* For each configuration of ASLR, this figure depicts the fraction of duplication observed in the memory contents of C_2 for different experimental configuration over various window sizes. The notations used for plotting the experimental configurations are same as mentioned above.
3. *Figure 3.4:* For each configuration of ASLR, this figure plots a bar graph comparing the fraction of duplication observed in the memory contents of C_1 and C_2 based on the placement (P) of the two containers relative to each other. The used for depicting an experimental configuration on the X axis is F_1F_2C . For example, PNC_1 implies that bar graph is plotted comparing the fraction of duplication observed in the memory contents of C_1 in an experimental configuration wherein C_1 and C_2 were executing functions dependent on Pandas and Numpy respectively.

Keeping the aforementioned notations in mind, we present the results of our experiments as follows:

[R1] More duplication is generally observed with smaller window sizes:

Figures 3.2a and 3.2b show the duplication observed in C_1 for various win-

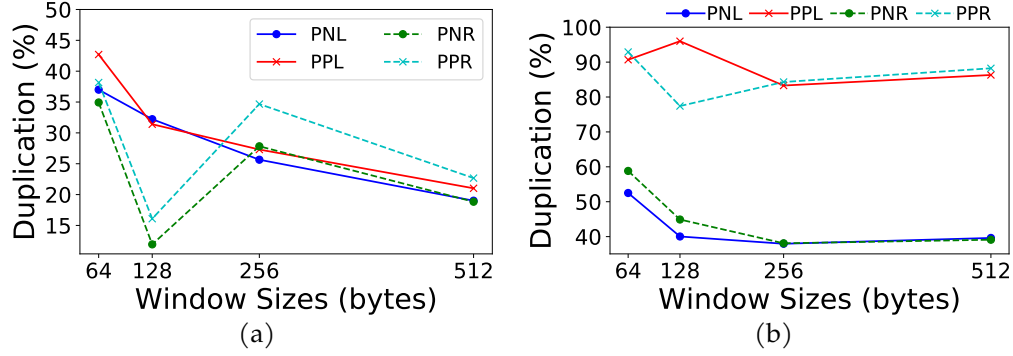


Figure 3.3: Duplication observed in C_2 for various window sizes with (a) ASLR enabled, (b) ASLR disabled

dow sizes with and without ASLR respectively. It can be seen that smaller window sizes have more duplication in both cases. This is intuitive (Appendix A.1) and similar to observations made for virtual machines in [21]. Similar observations can be drawn from Figures 3.3a and 3.3b for C_2 .

[R2] The duplication does not vary much based on the placement of the containers: Figures 3.2a and 3.2b show that changing the placement of C_1 with respect to C_2 only varies the duplication by 2-5%, with or without ASLR. Similar observations hold true from Figures 3.3a and 3.3b for C_2 and from Figures 3.4a and 3.4b for page level experiments.

[R3] Significantly high duplication is observed when ASLR is disabled: It can be observed from Figures 3.2a and 3.2b for C_1 that disabling ASLR increases the duplication observed by 55% in some cases. Similar observations hold true for C_2 from Figures 3.3a and 3.3b and for page level experiments from Figures 3.4a and 3.4b. Given that code is not user facing in serverless platforms, security vulnerabilities like code injection and heap disruption are not possible. Hence, we think that disabling ASLR for squeezing out maximum benefits from deduplication is a viable option.

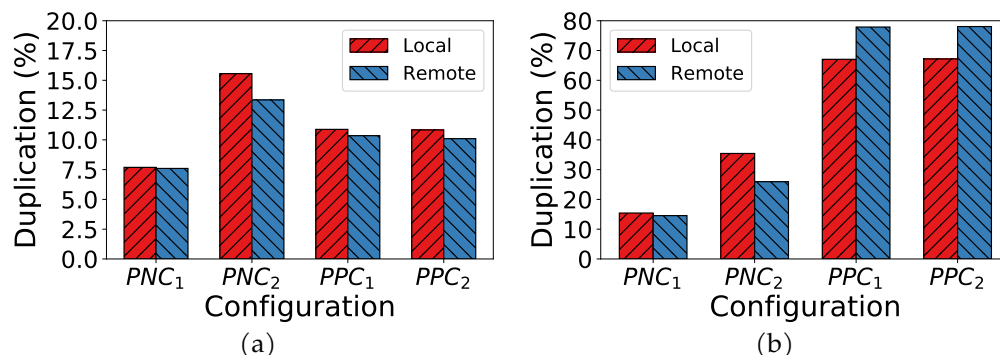


Figure 3.4: Page level (4096 bytes) duplication observed for different configurations of experimental parameters with (a) ASLR enabled, (b) ASLR disabled

[R4] With ASLR disabled, duplication observed is higher for containers having same dependencies: From Figures 3.2b, 3.3b and 3.4b, it can be observed that containers having same dependencies exhibit much higher duplication, with 60% more duplication in some cases. This is as per our expectations.

[R5] Significant amount of duplication exists currently in the memory contents of worker machines of serverless platforms today. Let's expand further in the example mentioned in Section 3.1 - let's say that F_1 depends on Pandas and F_2 depends on Numpy. We know from our experiments that the container environments of F_1 and F_2 have a memory consumption of $M_1 = 60\text{MB}$ and $M_2 = 45\text{MB}$ respectively. Furthermore, let's assume that $n = 1000$ invocations happen for F_1 and $m = 2000$ invocations happen for F_2 at a time instant t_1 . Finally, let's assume that we have disabled ASLR in our worker machine fleet since code injection and heap disruption attacks are not a vulnerability to the use case of serverless platforms. Based on our observations in regards to page level duplication between two instances of containers belonging to different functions having Pandas

and Numpy as their dependencies respectively (Figure 3.4b), we can assume $\alpha_1 = 0.15$ and $\alpha_2 = 0.28$. Putting these numbers into Equation 3.1, we get the duplication as 184095MB, which is 1.8GB. Indeed this is significant amount of duplication, which could potentially be avoided by only storing 105MB ($60 + 45$) amount of memory contents across the global memory pool of worker machines in the serverless system.

3.2.5 Future Work

Characterize and quantify duplication. Although our results do indicate existence of significant amount of duplication between the memory contents of container environments, it would be nice to perform more extensive experiments to further understand where exactly the duplication comes from. Some specific questions that would be ideal to answer are as follows:

1. *What is the characterization of duplication observed across data and code segments in the memory?* Since functions used in our experiments do not write any data, we expect most of the duplication to be from the code segments. However, it would be nice to formally quantify these numbers.
2. *What is the characterization of private and shared memory chunks among the memory chunks for which duplication is observed?*

Residual memory. We observe that in our experiments, 5-10% of the memory belonging to the containers does not get swapped out. Although the results still give us enough confidence on the existence of duplication, it would be nice to (i) figure out why residual memory does not get swapped out (ii) perform experiments to quantify the duplication between memory contents of container environments wherein no residual memory remains. One possibility to do this is by modifying the *Checkpoint/Restore in Userspace (CRIU)* [15] tool in Linux. The tool currently gets VM areas

from smaps [47] for the container, serializes them into protocol buffers [41] and writes the protocol buffers into disk files. A potential way to perform the experiments would be to intervene the tool after getting the VM areas from smaps and dumping them into a file.

4 DESIGN

4.1 Architecture

In this section, we discuss the architecture of a serverless system that performs deduplication of the memory contents of its global worker memory pool.

4.1.1 Load Balancer, Database and Scheduler

As shown in Figure 4.1, the *Load Balancer*, *Scheduler* and *Database* components of the serverless system do not change, and hence are grayed out in the diagram.

4.1.2 Deduplication Service

In order to deal with memory deduplication, we add a *deduplication service* to the serverless system having *dedup controller* and *dedup client* components as discussed below.

4.1.2.1 Dedup Controller

The controller stores metadata regarding the memory chunks currently available in the global worker memory pool of the serverless system. This is achieved by storing a *memory map* data structure in the controller which is essentially a map, wherein the key is the fingerprint (for example, md5 hash) of the memory chunk and the value is a list mentioning the worker machine where the chunk is stored, the memory location in the machine where the memory chunk is available, and the number of containers in the system currently in the deduplicated state which will need this chunk for reconstruction. The controller can consist of a cluster of machines for

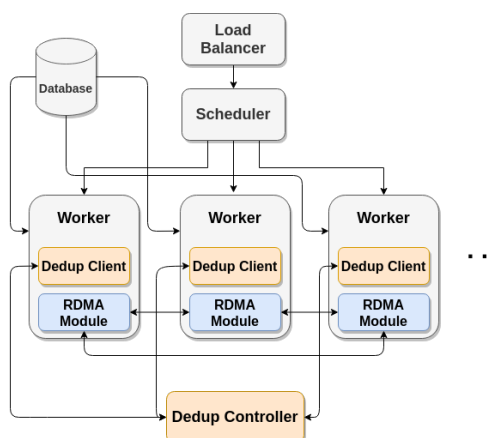


Figure 4.1: Serverless architecture for performing deduplication

fault tolerance, and the data structure can be kept synchronized using a consensus protocol (for example, Raft [39]).

4.1.2.2 Dedup Client

The dedup client runs at each worker machine in the worker pool of the serverless system. For each worker machine, the dedup client is responsible for coordinating with the dedup controller for performing memory deduplication of memory contents of containers in the machine. The dedup client stores a *container map*, wherein the key is the container ID and the value is a list of hashes of memory chunks that need to be fetched from remote memory to reconstruct the memory state of the corresponding container.

4.1.3 RDMA module

An RDMA module is added at each worker machine in the serverless system, and is tasked with establishing RDMA connections with all other worker machines in the system. This can be done during the startup of the system.

4.2 Workflow

4.2.1 Deduplicating a container

When the *keep warm time* (refer 2.1.2) of a container *C* has surpassed, it becomes a candidate for deduplication. The following events happen for deduplicating the container:

1. The dedup client breaks down the memory contents of the container into chunks of sizes configured for deduplication.
2. For each chunk, the client computes a hash and sends it over to the controller.
3. The controller acknowledges the presence/absence of the chunk in the global memory pool of the serverless system by consulting its memory map. If the chunk is present, the controller increases the count in its memory map for the number of deduplicated containers that need this chunk for restoration.
4. If the controller responds stating that the chunk is present in the global memory pool of the serverless system, the chunk is discarded from *C*.
5. If the controller responds stating that the chunk is absent in the global memory pool of the serverless system, the chunk is retained in *C*. The dedup client communicates to the controller to update its memory map accordingly.

4.2.2 Starting a deduplicated container

When a deduplicated container *C* needs to be started, the following events occur:

1. The dedup client retrieves the hashes of memory chunks discarded for the container *C* by consulting its container map.

2. The list of hashes thus obtained are sent to the dedup controller.
3. The dedup controller responds with the remote locations of worker machines containing memory chunks corresponding to the hashes sent by the dedup client.
4. The dedup client interacts with the RDMA module in the worker machine to fetch the memory chunks using the locations provided by the dedup controller.
5. The container is started since the memory state is restored.

4.2.3 Evicting a container

When a container C needs to be evicted, the following events occur:

1. The dedup client breaks down the memory contents of C into chunks of sizes configured for deduplication.
2. For each chunk, the client computes a hash and sends it over to the controller.
3. The controller retrieves the list corresponding to the chunk from its memory map and decreases the count of number of deduplicated containers in the system that need this chunk for restoration.
4. If the count drops to 0, the controller, sends acknowledgement to the client to discard the chunk.
5. If the count is greater than 0, the controller needs to find another machine in the system to migrate the chunk into - we leave this as future work.

5 RELATED WORK AND EVALUATION METHODOLOGY

5.1 Related Work

Serverless characterization: [45] looks at how network intensive applications run in serverless platforms whereas [30; 31; 42] characterize the storage requirements of serverless applications. In [53], a large measurement study was conducted to understand performance, resource management and isolation in serverless platforms. Similarly, [33] also conducted measurements on the public offerings of serverless frameworks. More recently, [43] characterize the entire production serverless workload of Azure functions and draw some similar observations as ones made from the characterization in our work.

Sandbox overhead reduction: [38] reduces the start up times of functions in OpenLambda [24] through caching python runtimes and packages, and uses low-latency isolation primitives. [11] advocates for the usage of language based isolation instead of using traditional virtualization techniques. [3] proposes a two level isolation wherein functions of the same application run within the same container as separate processes. [35] identifies that container networking setup takes significant time and pre-creates such resources to overcome the overhead, and dynamically binds to a container. More recently, [1] develops a lightweight Virtual Machine Monitor (VMM) specialized for serverless workloads. All these efforts are complementary to our work of reducing the impact of sandbox setup overheads.

Scheduling in serverless: [27] argues for cluster-level centralized and core-granular scheduler for serverless functions. The maintenance of global view of cluster resources reduces queue imbalances while the core

granularity reduces interference, thereby enabling reduced performance variability. [50] proposes a quality-of-service function scheduling and allocation framework which allows developers or administrators to easily define how serverless functions and applications should be deployed, capped, prioritized, or altered based on easily configured, flexible policies. [45] is a latency-sensitive serverless framework that scales scheduling, proactively creates sandboxes, and schedules functions with a shortest remaining slack first algorithm [23; 48]. Our work is complementary to these in that we aim to improve the latencies of serverless applications by reducing cold starts without changing the scheduler.

Decreasing memory footprint in serverless: [3] and [38] demonstrate both shortened deployment time and decreased memory footprints by forking execution from existing interpreter processes and applying Copy On Write (COW) sharing across their address spaces. For reduced start times, functions in [12] are deployed from unikernel snapshots, thereby bypassing expensive initialization steps. For reducing the memory footprints of snapshots, they apply page-level sharing across the entire software stack required to execute the functions. Our approach is different from these in that we identify existing duplication in the memory contents of sandboxes corresponding to functions belonging to a particular user. Subsequently, we plan to deduplicate the duplication thus found, resulting in an overall reduction in the memory consumption of serverless platforms.

5.2 Evaluation Methodology

Once implemented, the following metrics need to be collected and evaluated to quantify the efficacy of the system:

1. **Memory footprints:** For a given workload, the memory footprints are expected to decrease for our system since we do not store redun-

dant memory chunks in the memory pool of worker machines.

2. **Number of cold starts:** For a given workload, the number of cold starts are expected to decrease. A cold start would happen in our system if the dedup client figures out that the cost for reconstruction of memory chunks is higher than that of a cold start. Then, the system would only be efficient if that is not the case very often.
3. **Memory map lookup time:** We would need to ensure that the memory map data structure in the dedup controller has consistent metadata accessible to all worker machines in the serverless platform. We can use quorum or consensus protocol to achieve such consistency. However, we would need to evaluate the round trip latency for a dedup client to get a required metadata from the dedup controller to ensure that it is optimal for our use case.

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

Serverless computing platforms enable application developers to focus on the business logic, thereby moving scalability and deployment related concerns towards the provider of the serverless platform. Subsequently, the platform providers are faced with new resource management challenges. In this work, we have considered the specific problem of cold starts. We have experimentally shown that there exists redundancy (upto 85% in some cases between two container instances) among the memory contents of containers the cloud providers store in the warm state in their infrastructure. These currently exist since cloud providers today do not consider the global memory pool of the serverless infrastructure in a content aware manner. We have presented a system design to visualize the memory pool of worker machines in the serverless platform as one big unified memory, which can be accessed at low latencies by any worker using RDMA. Subsequently, the design focuses on ensuring that each unique memory chunk is only stored once in the unified memory pool. Doing so can help reduce the overall memory utilization and hence enable the cloud providers to keep more containers around in warm state, thereby reducing the number of cold starts and improving the overall performance of the system.

6.2 Future Work

Extensive experiments for characterizing duplication: As discussed in Section 3.2.5, it would be nice to perform more experiments to characterize the duplication between two containers and also to dig deeper into the current experiments.

System implementation: The system needs to be implemented to see if the expected benefits can be achieved. We think that the easiest way to implement the system would be in user space. Some parts of CRIU can be borrowed in the implementation since it also scans through the memory contents of the container instance which is being checkpointed.

A APPENDIX

A.1 Probability of duplication

Let's say that we have to find the probability of two strings (S_1 and S_2) of size s , wherein each character in the string belongs to the set $\{0, 1\}$. Then, S_1 and S_2 can take 2^s different values. Therefore, the combination of S_1 and S_2 can take $2^s * 2^s$, i.e 2^{2*s} different values. Out of these combinations, S_1 and S_2 will be exactly same in 2^s combinations. Therefore, the probability of duplication is given by $P = 2^s / 2^{2*s}$. On simplification, we get $P = 1/2^s$. Hence the probability of duplication decreases as the size increases.

BIBLIOGRAPHY

- [1] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)* (2020), pp. 419–434.
- [2] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 419–434.
- [3] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. {SAND}: Towards high-performance serverless computing. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 923–935.
- [4] Address Space Layout Randomization. <https://outflux.net/slides/2013/lss/kaslr.pdf>, 2020.
- [5] AWS Lambda. <https://aws.amazon.com/lambda/>, 2020.
- [6] AWS Redis. <https://aws.amazon.com/redis/>, 2020.
- [7] AWS S3. <https://aws.amazon.com/s3/>, 2020.
- [8] AWS Serverless Applications Repository. <https://aws.amazon.com/serverless/serverlessrepo/>, 2020.
- [9] Azure Functions. <https://azure.microsoft.com/en-us/services/functions/>, 2020.
- [10] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

- [11] Boucher, S., Kalia, A., Andersen, D. G., and Kaminsky, M. Putting the "micro" back in microservice. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 645–650.
- [12] Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., and Appavoo, J. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.
- [13] Callaghan, B., Lingutla-Raj, T., Chiu, A., Staubach, P., and Asad, O. Nfs over rdma. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications* (2003), pp. 196–208.
- [14] CloudLab. <https://www.cloudlab.us/>, 2020.
- [15] CRIU. https://www.criu.org/Main_Page, 2020.
- [16] Daw, N., Bellur, U., and Kulkarni, P. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference* (2020), pp. 356–370.
- [17] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)* (2014), pp. 401–414.
- [18] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*. Springer, 2017, pp. 195–216.
- [19] Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C., and Iosup, A. Serverless applications: Why, when, and how? *IEEE Software* (2020).
- [20] Google Cloud Functions. <https://cloud.google.com/functions>, 2020.
- [21] Gupta, D., Lee, S., Vrabie, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM* 53, 10 (2010), 85–93.

- [22] Guz, Z., Li, H., Shayesteh, A., and Balakrishnan, V. Performance characterization of nvme-over-fabrics storage disaggregation. *ACM Transactions on Storage (TOS)* 14, 4 (2018), 1–18.
- [23] Harchol-Balter, M., Schroeder, B., Bansal, N., and Agrawal, M. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems (TOCS)* 21, 2 (2003), 207–233.
- [24] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).
- [25] Islam, N. S., Wasi-ur Rahman, M., Lu, X., and Panda, D. K. High performance design for hdfs with byte-addressability of nvm and rdma. In *Proceedings of the 2016 International Conference on Supercomputing* (New York, NY, USA, 2016), ICS '16, Association for Computing Machinery.
- [26] Jiang, L., Pei, Y., and Zhao, J. Overview of serverless architecture research. In *Journal of Physics: Conference Series* (2020), vol. 1453, p. 012119.
- [27] Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 158–164.
- [28] Kalia, A., Kaminsky, M., and Andersen, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), pp. 295–306.
- [29] Kalia, A., Kaminsky, M., and Andersen, D. G. Design guidelines for high performance {RDMA} systems. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)* (2016), pp. 437–450.
- [30] Klimovic, A., Wang, Y., Kozyrakis, C., Stuedi, P., Pfefferle, J., and Trivedi, A. Understanding ephemeral storage for serverless analytics. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 789–794.

- [31] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 427–444.
- [32] LXC containers. <https://linuxcontainers.org/>, 2020.
- [33] McGrath, G., and Brenner, P. R. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (2017), IEEE, pp. 405–410.
- [34] Mitchell, C., Geng, Y., and Li, J. Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)* (2013), pp. 103–114.
- [35] Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. Agile cold starts for scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (2019).
- [36] Oakes, E., Yang, L., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (2017), pp. 395–400.
- [37] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 57–70.
- [38] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 57–70.
- [39] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 305–319.

- [40] OpenWhisk. <https://openwhisk.apache.org/>, 2020.
- [41] Protocol Buffers. <https://developers.google.com/protocol-buffers>, 2020.
- [42] Pu, Q., Venkataraman, S., and Stoica, I. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 193–206.
- [43] Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *arXiv preprint arXiv:2003.03423* (2020).
- [44] Shi, J., Yao, Y., Chen, R., Chen, H., and Li, F. Fast and concurrent {RDF} queries with rdma-based distributed graph exploration. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 317–332.
- [45] Singhvi, A., Banerjee, S., Harchol, Y., Akella, A., Peek, M., and Rydin, P. Granular computing and network intensive applications: Friends or foes? In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), pp. 157–163.
- [46] Slack. <https://slack.com/>, 2020.
- [47] smaps. <https://www.cpan.org/modules/by-module/Linux/Linux-Smaps-0.14.readme>, 2020.
- [48] Smith, D. R. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research* 26, 1 (1978), 197–199.
- [49] Spring, N. T., and Wetherall, D. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2000), SIGCOMM '00, Association for Computing Machinery, p. 87–95.

- [50] Tariq, A., Pahl, A., Nimmagadda, S., Rozner, E., and Lanka, S. Sequoia: enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020), pp. 311–327.
- [51] Vemula, R. A new era of serverless computing. In *Integrating Serverless Architecture*. Springer, 2019, pp. 1–22.
- [52] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 133–146.
- [53] Wang, L., Li, M., Zhang, Y., Ristenpart, T., and Swift, M. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 133–146.
- [54] Wang, S., Lou, C., Chen, R., and Chen, H. Fast and concurrent {RDF} queries using rdma-assisted {GPU} graph exploration. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 651–664.
- [55] Wu, J., Wyckoff, P., and Dhabaleswar Panda. Pvfs over infiniband: design and performance evaluation. In *2003 International Conference on Parallel Processing, 2003. Proceedings.* (2003), pp. 125–132.
- [56] Xue, J., Miao, Y., Chen, C., Wu, M., Zhang, L., and Zhou, L. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), pp. 1–14.