Funded by
the European Union

HORIZON EUROPE FRAMEWORK PROGRAMME

CLOUDSTARS

(Grant agreement No 101086248)

Cloud Open-Source Research Mobility Network

SECONDMENT REPORT

Coordinator: Universitat Rovira i Virgili

TABLE OF CONTENTS

GENERAL INFORMATION ON THE PROJECT

Name:	Aitor Arjona Pérez
Position:	PhD Student
Organization:	Universitat Rovira i Virgili
To:	IBM Research Zurich
Duration of Secondment (with date of arrival and departure):	90 days (1 April - 30 June)
Empresarial tutor:	Dr. Bernard Metzler
Tasks:	T3.3
WP:	WP3
Objectives of Secondment:	The objective includes, but is not limited to, studying current research challenges related to ephemeral data storage for large-scale scientific data analytics in the Cloud in the context of the GEDS open-source software project (https://github.com/IBM/GEDS) being currently developed at the host organization.
Skills and Knowledge Acquired:	Viable approaches for efficient data sharing on co-located containers in Kubernetes deployments.
Papers Published:	
Dissemination activities (mass media, presentations, talks, workshops, conferences):	Presented on Workshop on Serverless Computing Experience http://www.wikicfp.com/cfp/servlet/event.showcfp?eventid=174201&copyownerid=93722
Links to Published Works:	
Github repositories and project name:	https://github.com/aitorarjona/cloudstars-kubernetes-shared-volumes
Collaborations with other partners:	

DESCRIPTION OF THE TECHNICAL WORK

Towards efficient and secure data sharing between co-located serverless containers in Kubernetes

CLOUDSTARS EU Research Mobility Program Secondement Report

Aitor Arjona
Universitat Rovira i Virgili
Tarragona, Spain
aitor.arjona@urv.cat

ABSTRACT

The serverless computing paradigm is widely recognized as a practical solution for highly elastic compute and data-intensive workloads in the Cloud. Emerging serverless Cloud services based on container technologies provide higher degrees of flexibility for adopting new applications. However, managing temporary data in serverless environments remains a challenge. The stateless nature of serverless computing requires reliance on disaggregated storage, leading to latency issues due to data movements and performance degradation. Existing caching and temporary data store approaches present limitations in handling large datasets or impose additional infrastructure costs and management complexities.

To address these challenges, we propose the inclusion of GEDS (Generic Ephemeral Data Store) in serverless Kubernetes architectures. The objective is to leverage node locality using the ephemeral host file system allocated for each serverless container, enabling applications to effectively store and share temporary data across concurrent and successive invocations. This article explores the limitations and opportunities for effective data sharing between serverless co-located containers in Kubernetes deployments. Our findings demonstrate how by using memory-mapped files and file descriptor passing through shared volume mounts provides effective storage resources sharing between containers efficiently and securely.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;
• **Information systems → Distributed storage**.

KEYWORDS

ephemeral storage, serverless, Kubernetes, Cloud

1 INTRODUCTION

The serverless computing model, originally designed for lightweight web services with fluctuating demand, has undergone significant infrastructure improvements over the years that have significantly broadened its applicability. We are witnessing the emergence of new serverless services in public Clouds that go beyond the more constrained *Functions-as-a-Service* model, offering in contrast greater levels of flexibility [3, 12]. Some of them base their service on offering *serverless containers* on top of standard infrastructures such as Kubernetes. For example, IBM Cloud Code Engine [7] offers a fully-managed serverless service allowing users to deploy Kubernetes constructs, such as Deployments or Jobs. Similarly, Google Cloud Run [2] provides a serverless managed service for Knative deployments. These container-based services enable a broader range of workloads to be adopted and benefit from a serverless experience.

However, the serverless paradigm continues to face a well-known limitation in terms of managing temporary data. The stateless nature of serverless computing leads to a *data-shipping architecture*, where serverless applications must rely on remote disaggregated storage due to the inability to retain state or provide addressability between containers [14]. This limitation significantly hampers the adoption of a serverless approach for complex applications that involve large temporary data movements across stages of a workflow.

This work proposes to address the aforementioned challenge by enabling serverless containers to share storage resources across containers running on the same host but also within the entire compute cluster. To achieve this, we study the potential of leveraging GEDS (Generic Ephemeral Data Store) as storage system. GEDS is a storage middleware for ephemeral temporary data, with the goal of reducing the resource footprint of the storage system by running co-located within the compute cluster. The objective of this work is to examine the current design and architecture of GEDS and propose improvements to enhance its applicability for the effective sharing of storage resources in serverless multi-tenant scenarios. Our focus lies in exploring the opportunities for data sharing to leverage data locality in co-located but isolated containers on Kubernetes deployments for serverless workloads.

In summary, this work provides the following insights:

(1) We propose improvements to the GEDS architecture, with the goal of enabling data sharing exploiting node-locality. The suggested changes involve deploying GEDS as a *daemon* on each compute cluster node. Application containers communicate and coordinate with the system to access and share data between co-located and remote containers.
(2) We study and benchmark the performance of different mechanisms for shared volume mounts in co-located containers within the same Kubernetes worker nodes, namely `hostPath` volumes and CSI drives. Due to security concerns associated with Host Path volume mounts, we propose utilizing a custom CSI driver that mounts shared local volumes between containers of the same tenant on the same node.
(3) We conduct various experiments to evaluate the behavior of different mechanisms for data sharing in Kubernetes when using shared volume mounts. Specifically, we demonstrate how containers can utilize memory mapping backed by files stored in the shared volume as a secure approach to shared memory between containers. Read-only memory maps do not contribute to memory usage, allowing for efficient resource utilization for cached data. Additionally, we show how file descriptors can be safely passed to read and write data on resources allocated in different containers on the same node.

This document serves as a report for the research insights and activities conducted during the author's tenure in IBM Research Zurich laboratories in the months of April to June of 2023 as part of the Cloudstars researcher mobility program[1].

2 BACKGROUND AND RELATED WORK

Temporary data refers to data produced within workflow stages for sharing data between tasks — it is short-lived, it can be re-generated and it's only useful during the workflow execution. Temporary data management in serverless workloads has been extensively studied due to its impact on performance and scalability [15].

One commonly adopted solution for temporary data storage is the utilization of DRAM-based caching mechanisms within the serverless compute cluster [19, 22, 23]. Nevertheless, it is arguable whether these approaches effectively handle large data volumes, as memory-based stores can quickly overflow.

Alternatively, Klimovic et al. [16], Stuedi et al. [25] suggest employing a high-performance disaggregated ephemeral data storage solution for temporary data in serverless workloads. The rationale behind this approach is that temporary data stores can relax data durability guarantees and prioritize performance. In the event of a failure, the workflow can be re-executed to regenerate any data lost. By using a disaggregated approach, their storage system can accommodate large data volumes by allowing flexible scaling of both compute and storage resources independently. However, it also comes at the cost of additional infrastructure expenses and increased data latency due to remote access, requiring high-performance hardware that may not be readily available in standard Cloud environments.

Romero et al. [21] propose *Faa$T*, a transparent collaborative caching system that is deployed co-located within concurrent distributed instances of a serverless application. The aim of their approach is to reduce data access latency by intercepting calls to object storage. In *Faa$T*, data durability guarantee can be configured in order to achieve better performance for ephemeral data. However, the system's current design either persists all data or treats all data as ephemeral, which poses a challenge for scenarios where some data, such as output data, needs to be persisted while allowing temporary data to be ephemeral. A more granular approach is needed to address such mixed durability requirements effectively.

Merenstein et al. [18] introduce F3 in their recent publication. F3 enables to share ephemeral data among functions within a Functions-as-a-Service (FaaS) platform. Their system supports transparent integration, as applications interact with it through the standard POSIX file system interface. Contrary to *Faa$T*, F3 offers to selectively determine the persistence of data at both the file and directory levels. However, F3 relies on a network file system as a persistency layer, requiring additional management and allocation of resources for storage. Consequently, applications utilizing object storage for data access must employ other storage middleware to fulfill this requirement.

In summary, existing approaches from academia either fall short in supporting large temporary data, which is a crucial requirement for large-scale data-intensive workloads [15], or involve the deployment of additional non-serverless resources, which undermines the benefits of a serverless architecture in terms of reduced management burden. In contrast, GEDS addresses these limitations by providing efficient sharing of ephemeral data with a lower resource footprint. Our focus lies on exploiting locality for data sharing between containers running on the same node. Additionally, GEDS enables fine-grained control over data durability semantics, allowing for more flexible and tailored data management strategies.

[1] https://cloudstars.eu

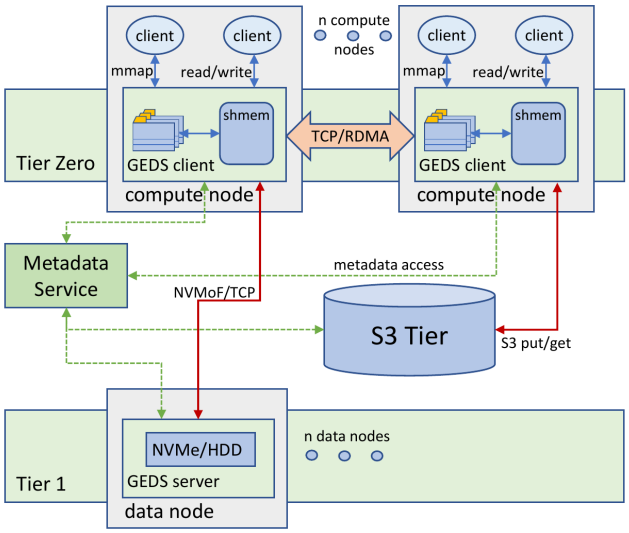

Figure 1: GEDS architecture diagram (© IBM)

3 GEDS: GENERIC EPHEMERAL DATA STORE

In this section, we introduce GEDS, the storage middleware being developed in the host organization at the time of writing. Figure 1 displays an architecture diagram of GEDS. GEDS is open source and is available in Github[2].

GEDS approach differs from related work by prioritizing the reduction of resource footprint dedicated to storage. In contrast to other approaches, GEDS runs co-located within the application code, loaded as a static or dynamic library. GEDS leverages hardware-accelerated network interconnect (RDMA) and storage (NVMe) available in the compute cluster nodes for fast and efficient access to warm temporary data. Moreover, GEDS implements a tiered architecture that allows to spill data to disaggregated lower tiers, thereby enabling it to accommodate larger data sets.

GEDS integrates with Cloud Object Storage, which serves two important purposes. Firstly, it enables reading input data through GEDS using file semantics, as the majority of data in the Cloud is typically available in Cloud Object Storage. This provides an opportunity to optimize data ingestion for serverless workloads, such as implementing caching mechanisms for frequently accessed or shared input data. Secondly, Cloud Object Storage is utilized as a persistency layer for storing output data. This ensures the durability of the processed data, allowing for subsequent access as needed, while also providing relaxed data persistency guarantees for temporary data under the same storage system. By leveraging Cloud Object Storage for both input and output data, GEDS leverages existing Cloud infrastructures and takes advantage of the scalability, reliability and serverless management offered by these services.

[2]https://github.com/IBM/GEDS

3.1 Improving GEDS to leverage node locality

The current implementation of GEDS involves embedding the GEDS client as a static or dynamic linked library within the application code. However, this design decision becomes a limiting factor when attempting to share storage resources managed by GEDS among multiple processes or containers on the same machine. GEDS lacks the capability to efficiently share data between co-located clients. Even when the data is already available on the same machine, co-located GEDS clients still rely on the metadata service to obtain the data location and establish a TCP connection over the published network interface to access it.

In this section, we propose architectural changes to GEDS with the objective to make it more suitable for resource sharing in the same node, between unrelated processes, and consequently, between containers.

The proposed architecture is illustrated in Figure 2. The primary design change involves decoupling GEDS from the application container and running it as a separate background process, commonly known as a *daemon*. To establish communication with the GEDS *daemon*, the application can employing a lightweight GEDS client and IPC (Inter-Process Communication) mechanisms such as UNIX pipes or sockets. By adopting this approach, different unrelated processes running on the same machine can make use of a shared GEDS client instance.

It is crucial to enable zero-copy direct access to data generated by one process from other data consumer processes, in order to avoid passing and copying data between the GEDS daemon process and application processes through IPC. This can be achieved by enabling the application direct access to the underlying storage media (such as memory or disk), or direct access to GEDS mechanisms to retrieve data from remote nodes if necessary (network). The primary responsibility of the GEDS daemon is then to handle synchronization concerns. Applications utilizing GEDS should be able to access data seamlessly through a unified interface, regardless of whether the data is locally shared or located remotely.

Before implementing a definitive solution, we first need a deep understanding of the limitations associated with data sharing in the targeted environment, which in this case is a Kubernetes cluster deployment. Specifically, we aim to assess how storage resources can be efficiently shared between Kubernetes Pods and identify the challenges and limitations involved. The work described focuses on providing insights

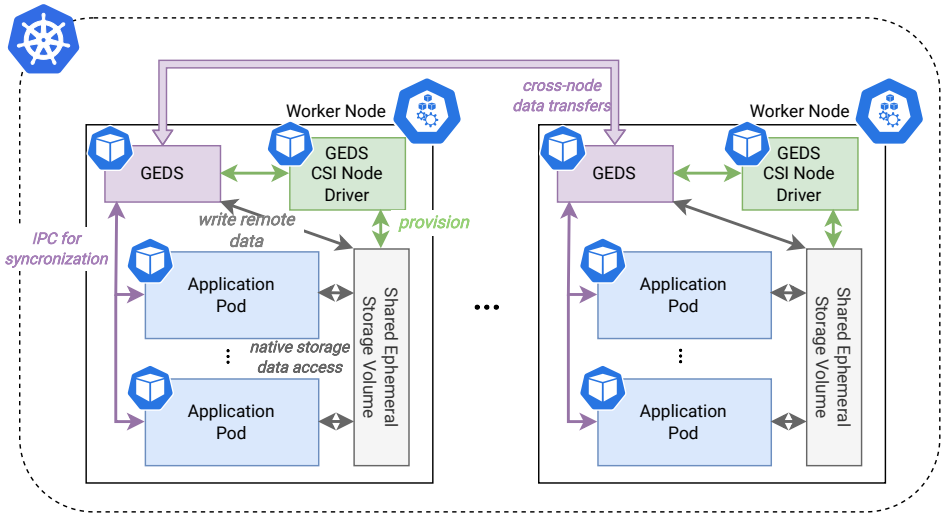

Figure 2: Proposed GEDS architecture to leverage locality

into this issue rather than proposing a specific implementation. The actual adoption of the proposed architecture is left as potential future work, which can be explored by future visiting researchers at the host organization.

At a first glance, we can differentiate three main challenges to be addressed in this approach:

(1) **Security and isolation:** Since we want to share data between containers, we are potentially breaking container boundaries. In multi-tenant Cloud environments, where physical nodes are shared by different tenants, it becomes crucial to maintain resource isolation and data security between containers of different tenants while allowing resource sharing among containers of the same tenant.
(2) **Performance:** A reasonable balance between security/isolation guarantees and performance is key in realistic multi-tenant Cloud scenarios. Our solution must find a compromise between security while ensuring reasonable performance.
(3) **Elasiticity:** Serverless workloads are characterized by high elasticity, where containers can be rapidly allocated and deallocated. To accommodate serverless workloads, the storage system must be capable of scaling out and in accordingly, efficiently handling the dynamic nature of serverless resource demands.

The following sections provide insight into how each of the challenges above can be addressed when adopting the proposed GEDS architecture.

4 SHARING STORAGE RESOURCES BETWEEN KUBERNETES PODS

Efficient and secure sharing of host resources is crucial in order to effectively leverage locality for co-located containers on the same node. Our focus lies in efficiently sharing resources for ephemeral storage, particularly for data-intensive applications. Specifically, we aim to share storage resources such as files, volumes, and file descriptors. In this regard, cross-process synchronization and other Inter-Process Communication (IPC) mechanisms like queues, locks, atomic shared objects, or semaphores are out of scope of this work. This section provides insight and discussion on various alternatives available in Kubernetes that can serve to accomplish our goal. All experiments and source code is publicly available in Github[3].

4.1 The “infamous” Host Path volume mounts

In a nutshell, the main objective is to enable efficient data sharing between co-located Pods, leveraging locality. However, achieving this goal presents a challenge due to Kubernetes design philosophy, which prioritizes scalability and high availability by treating stateless resources as ephemeral and replaceable. For instance, worker nodes in Kubernetes are replaceable, which means that resources from a failed node are redistributed to the remaining healthy worker nodes. This means that assigning Pods to specific nodes to use some data from drives located in it, although possible, is considered a bad practice. As a result, Kubernetes relies on disaggregated

[3]https://github.com/aitorarjona/cloudstars-kubernetes-shared-volumes

storage mechanisms for data persistency, such as Network File Systems (NFS).

Nevertheless, Kubernetes does offer the option to expose subdirectories of the underlying host node to its running containers trough Host Path volume mounts [13]. However, this approach presents several inconveniences. Firstly, the user needs to be aware of the structure of volumes and file systems available on the host, as the mounted host path must be specified in the Pod manifest. Moreover, the use of `hostPath` volume mounts has a notorious history of significant vulnerability issues that should not be ignored [20]. A vulnerability was found (`CVE-2017-1002101`) which exploited a symlink race condition, allowing the Pod to mount the root file system with root access inherited from the privileged kubelet permissions. Interestingly, although a security patch was applied, it resulted in another vulnerability with similar implications (`CVE-2021-25741`). It is important to highlight the official Kubernetes documentation's perspective on `hostPath` volumes [13]:

> *HostPath volumes present many security risks, and it is a best practice to avoid the use of HostPaths when possible. When a HostPath volume must be used, it should be scoped to only the required file or directory, and mounted as ReadOnly.*

In conclusion, while `hostPath` volumes offer an effective solution for accessing and sharing storage host resources in self-managed or on-premises Kubernetes deployments through volume mount bindings, their multiple security concerns make them unsuitable for a multi-tenant Cloud scenario.

4.2 Container Storage Interface

The Container Storage Interface (CSI) in Kubernetes is provided as a mechanism that enables third-party vendors to implement and extend new Kubernetes Storage Classes, thereby harnessing a diverse range of storage technologies for use within Kubernetes [27]. To accomplish this, vendors are required to implement a standard interface that incorporates the necessary logic for interacting with the desired storage system.

The CSI controller driver is responsible for the control logic when new Kubernetes volumes are created. It interacts with the underlying storage system to provision the required storage or managing permissions. For instance, a NFS CSI driver might create a new file system on the NFS server when a volume is created. This component must be unique within the cluster, although the driver can implement replication with a leader election protocol for increased availability in case of failure.

The CSI node driver handles the actual setup of storage on the host node where the Pod is scheduled. Following the NFS example, the CSI node driver would perform the actual `mount` operation of the remote NFS at a specific path to make it accessible from within the container. This component is deployed as a `daemonset` and runs on all worker nodes.

Both components must implement a gRPC service and respond to RPC calls to perform actions. The CSI framework includes additional components like the snapshotter controller, but for simplicity, they are not discussed here.

This approach is better suited for our objectives. Firstly, it ensures a clear separation of concerns between users and the logic of mounting ephemeral storage volumes across co-located containers. In this regard, users only need to specify the desired ephemeral volume class and its total capacity, leaving the CSI driver responsible for handling all the necessary setup and making the storage available to the containers.

Secondly, the CSI driver has the capability to still implement logic that enables the sharing of storage resources among containers belonging to the same tenant via mount bindings. In essence, a CSI driver implementing volume mount bindings from the host is virtually equivalent to `hostPath` volume mounts. This effectively fulfills the our requirement of sharing storage resources between containers.

Lastly, but most importantly, this approach addresses all security concerns associated with the `hostPath` volume mounts. The risk of a potential chroot jail break, as seen with hostPath mounts, is lower since the volume binding logic is handled by the CSI driver rather than within the application Pod itself.

We conclude that a custom CSI driver for GEDS implementing volume mount bindings presents a reasonable compromise between security (ensured by the CSI driver) and performance (achieved through native storage resource access via volume mounts).

4.3 Ephemeral storage performance

After discussion, we conducted a benchmark to assess the performance of the different ephemeral storage options available. The benchmark involves reading and writing a file to the ephemeral volume using a single process within a running Pod. We compared three options:

(i) Ephemeral `emptyDir` volume backed by the host root file system [9]. (ii) `hostPath` volume mounted on the temporary file system on the host (`/tmp`). (iii) CSI driver implementing a `hostPath` volume mount. Specifically, we used the official CSI driver for hostPath [4]. Although this driver is a "mock" implementation for CI/CD testing, it effectively provides

[4] https://github.com/kubernetes-csi/csi-driver-host-path

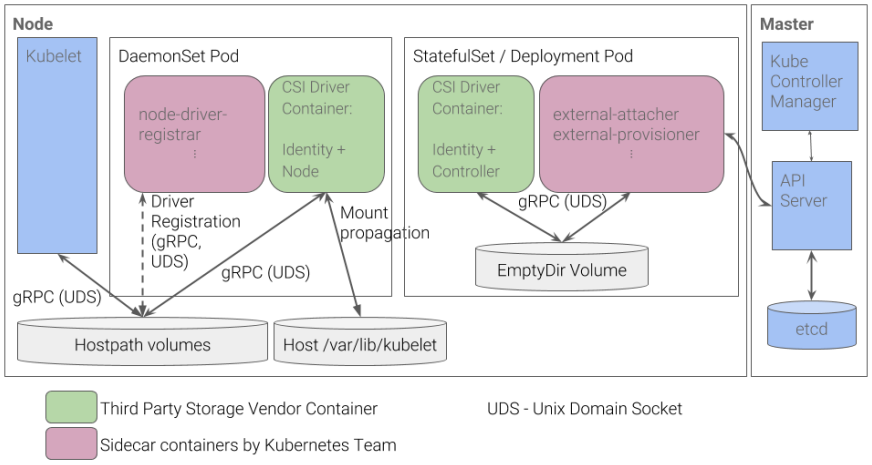

Figure 3: CSI driver architecture diagram

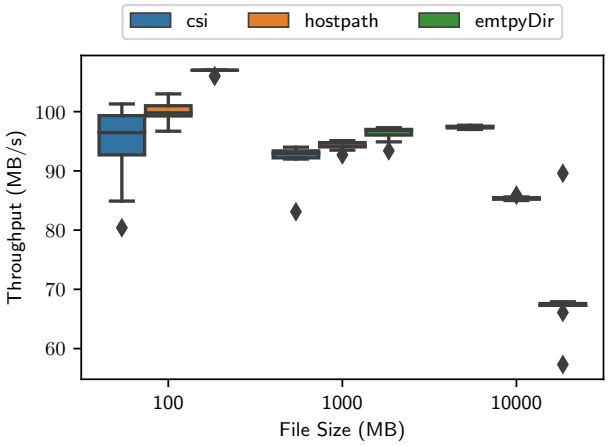

Figure 4: Ephemeral storage comparison read benchmark

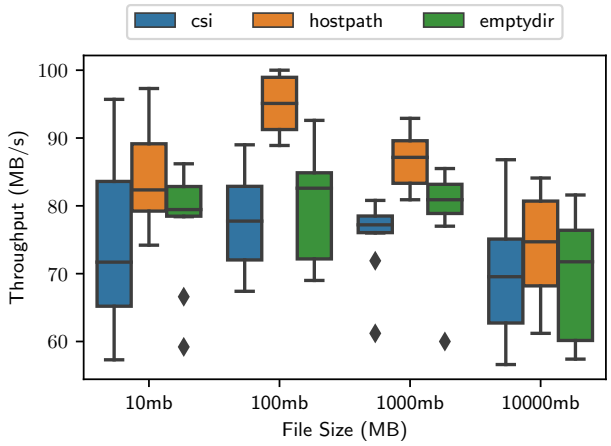

Figure 5: Ephemeral storage comparison write benchmark

functionality similar to hostPath volume mounts by implementing a wrapper over volume mounts using CSI.

To conduct the benchmark, we used `dd` configured with a block size of 2KB (`count`) and specified the total number of blocks to be read/written accordingly (`repeat`). We used `oflag=direct` and `iflag=direct` in `dd` to bypass write and read cache, respectively. The underlying storage medium used in the benchmark was a spinning disk, mounted as the root file system on the host node. These experiments are run on the infrastructure for research available at the host premises.

Figures 4 and 4 show the read and write results, respectively. The three options yield similar results for both reading and writing. However, for large files, the CSI driver provides better throughput. This baseline comparison with `emptyDir` ephemeral volume types will be useful for assessing the performance overhead of the final implementation, in order to further optimize the system.

Similarly, we wanted to compare an `emptyDir` volume backed by memory with an `emptyDir` volume backed by a high-performance NVMe drive. The objective was to assess the performance of backing data to disk using NVMe drives in tier zero. We used the `dd` command-line tool to perform the experiment, with the same conditions as above.

Figures 6 and 7 show the read and write results of this experiment, respectively. We can observe that for writing a large file, the throughput is comparable. However, reading from disk is almost half slower than from memory. These

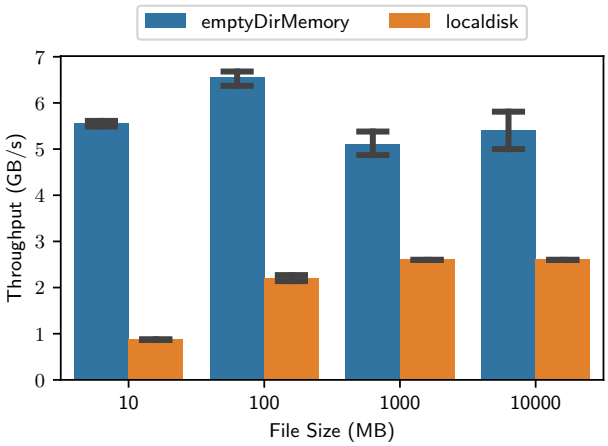

Figure 6: `emptyDir` ephemeral storage read benchmark

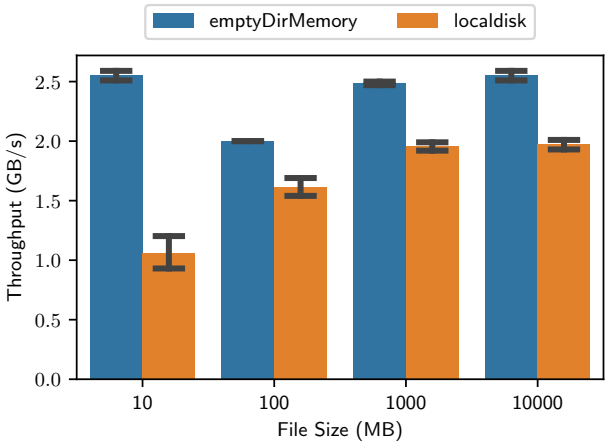

Figure 7: `emptyDir` ephemeral storage write benchmark

results can vary greatly when using caching, which was disabled for this experiment to assess the raw performance. Nonetheless, these results indicate that spilling data to disk when memory is full do not present significant performance degradation if using high-performance disks.

4.4 Memory mapping files in shared volume mounts

After discussing the viability of effectively and securely sharing host volumes across multiple containers, we want to explore the potential benefits it offers.

One such possibility is the Linux kernel's capability to memory-map files. In the Linux kernel, it is possible to map a kernel address space to a user address space, which is what

```c
int fd = open("/tmp/shared_file.bin",
    O_CREAT | O_RDWR, S_IRUSR |
    S_IWUSR | S_IRGRP | S_IWGRP);
ftruncate(fd, BUFFER_SIZE);
char *mmap_ptr = mmap(nullptr, BUFFER_SIZE,
    PROT_WRITE, MAP_SHARED, fd, 0);

memset(mmap_ptr, '\0', BUFFER_SIZE);
strcpy(mmap_ptr, "Hello world!");

printf("%s", mmap_ptr);
printf("Press enter to continue...");
getchar();
printf("%s", mmap_ptr);

munmap(mmap_ptr, BUFFER_SIZE);
close(fd);
```

Listing 1: Container `A` code.

we call *memory mapping* [10]. This is beneficial for performance as it eliminates copying data back and forth between the kernel and user memory space, reducing memory access penalties and overheads.

Memory mappings that are backed by a file allow users to read and write the file contents as if it were a contiguous section of memory, instead of using common file operations like `seek` or `read`. The kernel cache memory pages, which may be unordered, are directly mapped onto the virtual memory of the user process in the correct order. This allows user processes to read data directly from the kernel memory pages, avoiding to copy data to the user's process memory space. The required memory pages are loaded from the file automatically and lazily. This type of mapping is commonly referred to as a file-backed mapping or memory-mapped file.

When multiple processes map the same region of a file with the `MAP_SHARED` flag enabled [10], the physical memory pages are shared among them. In the case of shared memory mappings, modifications to the mapping's contents are instantly visible to other processes sharing the same mapping and are also reflected in the underlying file.

In this regard, we want to assess the behavior of shared memory maps between different isolated containers that share a common volume mount binding.

First, we run a simple experiment to check if two containers can share data by memory mapping the same file. For it, we run two Pods mounting the same `hostPath` subdirectory on `/tmp` and we allocate them on the same node using the `nodeSelector` tag on the Pod manifest. As seen in Section 4.2, `hostPath` mounts use volume mount bindings, which could also be possible using a CSI driver.

```
int fd = open("/tmp/shared_file.bin",
  O_RDWR, S_IRUSR | S_IWUSR
  | S_IRGRP | S_IWGRP);
char *mmap_ptr = mmap(nullptr, BUFFER_SIZE,
      PROT_WRITE, MAP_SHARED, fd, 0);

printf("%s", mmap_ptr);
strcpy(mmap_ptr, "Goodbye World!\0");

munmap(mmap_ptr, BUFFER_SIZE);
close(fd);
```

Listing 2: Container B code.

First, container A (Listing 1) opens and creates a file in the shared volume, we run `ftruncate` to allocate the file size on disk. Next, we proceed to create a memory map backed by that file, note the `MAP_SHARED` flag. Then, we copy the string `"Hello World!"` at the beginning of the buffer. On the other hand, container B (Listing 2) also opens the same file and memory-maps it, also using the `MAP_SHARED` flag. Container B prints the buffer contents, `"Hello World!"`. Next, we modify the buffer. From Container A, we print the buffer content, and now see `"Goodbye World!"`. Finally, both containers unmap and close the file. From this simple experiment, we can verify that both containers shared the same memory pages backed by the file mounted on the shared volume.

Next, we want to examine how memory is accounted for when using memory maps of a shared file between containers. In Kubernetes with `containerd`, resource constraints are managed by `cgroups` [1]. For this experiment, we use a similar setting as before. In this scenario, Container A will perform a `memset` syscall to populate the memory-mapped buffer. This ensures that this container "touches" all memory pages that correspond to the memory map. On the other hand, Container B will perform a `md5` checksum calculation on the buffer, reading its entire content iteratively. To monitor memory usage, we retrieve the value from a `cgroups` virtual file [5] located within the container.

Figure 8 shows the `cgroups` memory usage in bytes of both containers. "Writer process" corresponds to Container A, responsible for allocating and populating the memory map, while "Reader process" corresponds to Container B, which performs the md5 `checksum`. We see that the memory map size (512*MB*) is allocated to the Container A when the `memset` operation is called. However, Container A can read the buffer without requiring additional memory allocation for the memory-mapped pages.

Furthermore, we want to assess whether `cgroups` correctly manages exceeding the allocated resources when using

[5] `/sys/fs/cgroup/memory/memory.usage_in_bytes`

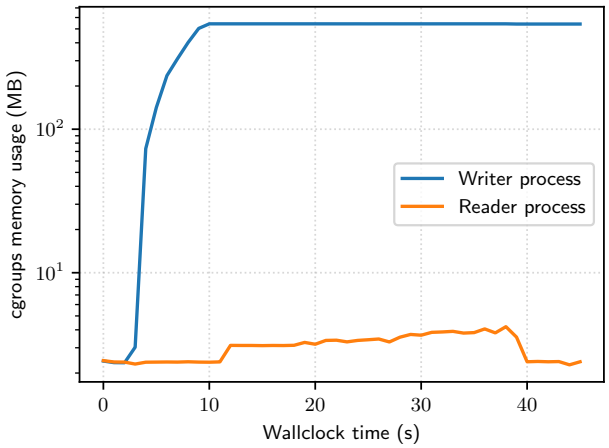

Figure 8: `cgroups` memory usage in bytes of two containers which write and read to a shared memory-mapped file buffer, respectively.

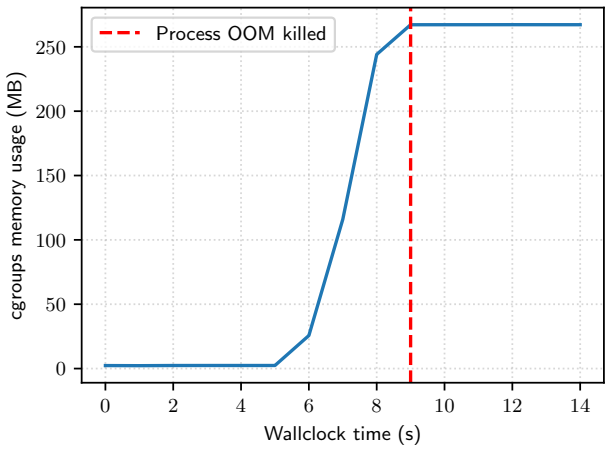

Figure 9: `cgroups` memory usage in bytes and OOM error of a container which writes to a memory-mapped file buffer.

memory-mapped files. To investigate this, we repeated the experiment with a container that had a memory limit of 256*MB*, while utilizing a 512*MB* memory-mapped file. Figure 9 shows the result. Indeed, `cgroups` kills the process with an out-of-memory error at the `memset` operation.

This finding is particularly noteworthy for write-once, read-many data scenarios. A "main" container can allocate read-only shared cached data on a shared memory map. Subsequently, multiple "worker" containers can access and read that data without incurring additional resource allocation,

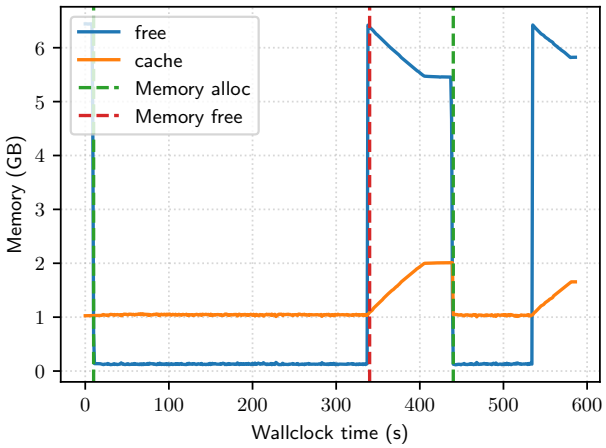

Figure 10: Operating system memory usage with a memory-mapped in a memory constrained scenario.

resulting in a more memory-efficient approach to sharing data among co-located Kubernetes containers.

Finally, we want to investigate the allocation of memory at the host level when utilizing shared memory-mapped files between containers. For this experiment, we set up a minikube [11] Kubernetes cluster with the virtualbox driver, configured with 6*GB* of RAM for the virtual machine. In this case, the shared memory map backed by the file at the shared volume has a size of 10*GB*. We generated this file using `dd`, copying data from `/dev/zero`. We run a resource-constrained Pod (`md5-pod`), with a maximum of 512*MB* of memory. This Pod memory-maps the aforementioned large file and performs a `md5` checksum on the buffer. In parallel, we launch another Pod (`mem-pod`) which allocated 5*GB* of memory by performing multiple `malloc` and `memset` calls. As a result, only 1*GB* of free memory remained for the host operating system and other Pods on the minikube node. We measure the memory consumption using `vmstat` tool.

Figure 10 presents the result of this experiment. We can see that the remaining 1*GB* of memory is allocated as cache memory by the kernel, rather than being occupied by user processes. In this case, the allocated cache memory is utilized by the `md5-pod` during its computation of the md5 checksum over the large buffer, leaving little to no free memory for the operating system. Then, we terminate the `mem-pod`, freeing all allocated memory to this container. This results in the cache memory usage increasing to 2*GB*, even though there is nearly 5*GB* of unused free memory. Next, we relaunch the `mem-pod` container, consuming again all of the free memory and a portion of the cache memory, leaving only 1*GB* of memory available for caching.

With this experiment, we observe that memory-mapped files are indeed accounted as cache memory by the kernel. Notably, there appears to be a limitation on the amount of memory that can be allocated for caching, warranting further investigation to determine potential limitations. Additionally, it is worth mentioning that we were able to memory-map a file significantly larger than the available physical memory on the node. In such cases, the kernel handles swapping the pages of the file-backed map accordingly.

4.5 Sharing file descriptors

An intriguing capability when sharing volume mount binds between containers involves passing file descriptors opened in one container and read or write from them in another container.

This functionality is already possible between unrelated processes on Linux systems through the use of Unix sockets [5, 8]. By using the `sendmsg` and `recvmsg` system calls with the `SCM_RIGHTS` flag enabled on the sent message, we can set an array of opened file descriptors as the message payload and have them duplicated on the receiving process. The Linux kernel processes messages with the `SCM_RIGHTS` flag by creating the appropriate file descriptors in the receiving process file descriptor table, which will point to the corresponding underlying opened device at the kernel. Semantically, this operation is equivalent to `dup` [4], which creates a duplicate of a file descriptor, returning a different file descriptor table entry. In this sense, both file descriptors share the underlying status, buffer, and offset. This means that reading any amount of bytes on one process will move the offset of the duplicated file descriptor the same amount. However, if this behavior is not desired, the sending process may close its file descriptor, making the duplicated one still available, and reopen the resource (e.g. file) if necessary.

Remarkably, containers sharing a common volume mount bind can achieve the very same behavior by placing the Unix named socket on the shared volume. To evaluate this behavior, we conducted a simple experiment adapted from [24], which involves two Pods, *sender* and *receiver*, deployed on the same node. The code for the sender and receiver Pods is listed in Listings 3 and 4, respectively. The Pods were configured to mount the same `hostPath` volume. The sender Pod creates a named Unix pipe within the shared volume. It also opens a file, located on the private ephemeral volume allocated for the container (`/tmp`). Then, it sends the file descriptor using `sendmsg` to the named socket using the `SCM_RIGHTS` flag. Simultaneously, the receiver Pod establishes a connection to the Unix socket and calls `recvmsg` to receive the file descriptor. It then proceeds to read data from the file descriptor, print it, and write some data back to the beginning of the buffer. The sender process can then seek back

```c
// Setup socket for sending messages
sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
strcpy(servaddr.sun_path, socket_path);
connect(sockfd, (struct sockaddr *)&servaddr,
  sizeof(servaddr));

// Open a local private file and send
// the fd trough sendmsg with SCM_RIGHTS flag
fd = open(file_path, O_RDWR);
send_fd(sockfd, "", 1, fd);

// Wait for receiving process to modify
// the file and read the result
printf("Press enter to continue...\n");
getchar();
lseek(fd, 0, SEEK_SET);
read(fd, buf, 12);
printf("%s", buf);
```

Listing 3: Sender process code.

```c
// Setup socket for receiving messages
unlink(sockpath);
listenfd = socket(AF_LOCAL, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_LOCAL;
strcpy(servaddr.sun_path, socket_path);
bind(listenfd, (struct sockaddr *)&servaddr,
  sizeof(servaddr));
listen(listenfd, 5);
clilen = sizeof(cliaddr);
connfd = accept(listenfd,
  (struct sockaddr *)&cliaddr, &clilen);

// Call recvmsg and set received fd
recv_fd(connfd, buf, 1, &fd);

// Read and write to the received fd
read(fd, buf, 13);
printf("%s", buf);
lseek(fd, 0, SEEK_SET);
write(fd, "Hola Mundo!\0", 12);
```

Listing 4: Receiver process code.

the offset and read the data written by the receiver process. Note that the opened file is located on the private ephemeral volume allocated for the sender Pod, which is not visible for the receiver Pod.

Through this experiment, we observed that a Pod effectively gained access to a file located within the isolated container of another co-located Pod by leveraging file descriptor sharing via Unix sockets on a shared volume mount. This approach provides a viable avenue for secure and efficient resource sharing between containers.

4.6 Discussion on file system access and FUSE

In this section, we will address the challenges associated with providing a file system interface for GEDS.

Many file systems from industry and academia are implemented trough FUSE [17]. FUSE [6], or File System in User Space, is a Linux framework to implement virtual file systems outside of the kernel, in user space. To support FUSE, the storage system needs to implement the FUSE interface, which includes functions for mounting and unmounting file systems, as well as operations on files and directories such as opening files, reading data, seeking, etc.

By utilizing FUSE as an adapter, applications can transparently access alternative storage systems that may do not provide native file system access. A notable example is S3FUSE, a commonly used FUSE file system that allows accessing S3 data as if it were a file system. Operations like opening, seeking, writing, and reading are translated into corresponding RESTful operations over the underlying object storage system.

Currently, GEDS implements a file-like interface, which needs to be used explicitly. One approach is to implement a FUSE interface for GEDS. However, providing a POSIX file system interface for an ephemeral data store like GEDS has significant implications. Applications that access data through a file system interface generally expect the data to be persisted as if it was being read from and written to a local disk. Yet, this does not align with the nature of ephemeral data storage systems, where data persistency is not enforced in order to prioritize performance. Although FUSE provides enough flexibility to not require a complete POSIX compliance, applications utilizing the ephemeral FUSE file system must adjust their usage accordingly and not expect a persistent file system, defeating in part the purpose of transparency.

GEDS already addresses this challenge by integrating both persistent and ephemeral data under a unified system interface. In the native API, the user explicitly specifies data persistency using prefixes such as `geds://` for ephemeral data or `s3://` for persisted data on object storage. However, with file system access, applications may store data in arbitrary locations without explicitly indicating its persistence. Some propose using special sub-directories or employing specific file open flags to indicate persistency on file system operations [18]. Nonetheless, these approaches also defeat the promise of backwards compatibility for legacy applications.

Even seemingly minor changes can require in-depth knowledge of the application internals to ensure proper handling of data persistency.

Given the considerations mentioned, we contend that file system access can be considered an additional feature of GEDS to accommodate a wider range of use cases. However, it should not be the primary system interface. To fully harness the potential of the system, applications should utilize the native GEDS interface, and explicitly indicate the data persistency as required. This approach not only helps to avoid potential overhead penalties associated with FUSE [26], but also ensures correctness on data persistency.

5 CONCLUSION

In this report, we have explored how GEDS can be leveraged to benefit from locality by sharing storage resources among co-located containers.

Specifically, we have seen how, by securing a shared volume on co-located containers, we enable efficient data handling operations such as memory mapping and passing file descriptors between isolated containers. Our approach promotes native storage access efficiency, minimizing unnecessary data copies and aiming for secure zero-copy data access.

These novel approaches have substantial implications for emerging serverless Cloud services based on container technologies and standard infrastructures like Kubernetes. By leveraging GEDS, the performance of data-intensive applications deployed on these platforms can be significantly boosted, leading to improved overall efficiency and scalability.

Future work directions: We now indicate future work for secondments carried out on the host organization can address some of the issues introduced in this report. (i) Complete a prototype integration of GEDS with the proposed architecture. (ii) Investigate the viability and implications of sharing a cache between different clients. We can assume a GEDS deployment per tenant, although this could benefit security, it also introduces repeated services per node. (iii) Investigate Kubernetes scheduling to pack on one node as many containers as possible, in order to further exploit locality benefits discussed in this report.

ACKNOWLEDGMENTS

I would like to express my gratitude to my hosts at IBM Zurich, Bernard Metzler, Pascal Spörri, and Radu Stoica, for their warm hospitality and unwavering support. I also thank my fellow project colleagues, Luis Veiga, Pezhman Nasirifard, and René Schwermerene, as well as the entire Cloudstars Research Mobility Program partners for their collaborative efforts.

This work has been carried out in the context of the Cloud Open Source Research Mobility Network CLOUDSTARS (grant number 101086248), which is funded by the Marie Skłodowska-Curie Actions (MSCA) EU program. Aitor Arjona is a URV Martí Franquès grant fellow.

REFERENCES

[1] [n. d.]. cgroups(7) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man7/cgroups.7.html. [Accessed 17-Jul-2023].

[2] [n. d.]. Cloud Run: Container to production in seconds | Google Cloud — cloud.google.com. https://cloud.google.com/run. [Accessed 18-Jul-2023].

[3] [n. d.]. Dataproc Serverless Google Codelabs — codelabs.developers.google.com. https://codelabs.developers.google.com/dataproc-serverless. [Accessed 25-07-2023].

[4] [n. d.]. dup(2) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man2/dup.2.html. [Accessed 18-Jul-2023].

[5] [n. d.]. fd-passing — keithp.com. https://keithp.com/blogs/fd-passing/. [Accessed 19-Jul-2023].

[6] [n. d.]. FUSE The Linux Kernel documentation. https://www.kernel.org/doc/html/next/filesystems/fuse.html. [Accessed 17-Jul-2023].

[7] [n. d.]. IBM Cloud Code Engine - IBM. https://www.ibm.com/cloud/code-engine. (Accessed on 07/13/2023).

[8] [n. d.]. Know your SCM_RIGHTS — blog.cloudflare.com. https://blog.cloudflare.com/know-your-scm_rights. [Accessed 19-Jul-2023].

[9] [n. d.]. Kubernetes Volumes. https://kubernetes.io/es/docs/concepts/storage/volumes/#emptydir. [Accessed 15-Jul-2023].

[10] [n. d.]. Memory mapping — The Linux Kernel documentation. https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html. (Accessed on 06/29/2023).

[11] [n. d.]. minikube start — minikube.sigs.k8s.io. https://minikube.sigs.k8s.io/docs/start/. [Accessed 17-Jul-2023].

[12] [n. d.]. Open Source Big Data Analytics Amazon EMR Serverless — Amazon Web Services — aws.amazon.com. https://aws.amazon.com/emr/serverless/. [Accessed 25-07-2023].

[13] [n. d.]. Volumes — kubernetes.io. https://kubernetes.io/docs/concepts/storage/volumes. [Accessed 14-Jul-2023].

[14] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. arXiv:1812.03651 [cs.DC]

[15] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[16] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[17] Kunal Lillaney, Vasily Tarasov, David Pease, and Randal Burns. 2019. Agni: An Efficient Dual-Access File System over Object Storage. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 390–402. https://doi.org/10.1145/3357223.3362703

[18] Alex Merenstein, Vasily Tarasov, Ali Anwar, Scott Guthridge, and Erez Zadok. 2023. F3: Serving Files Efficiently in Serverless Computing. In

Proceedings of the 16th ACM International Conference on Systems and Storage (Haifa, Israel) *(SYSTOR '23)*. Association for Computing Machinery, New York, NY, USA, 8–21. https://doi.org/10.1145/3579370.3594771

[19] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244. https://doi.org/10.1145/3447786.3456239

[20] Fred Raynal. [n. d.]. Kubernetes and HostPath, a Love-Hate Relationship — blog.quarkslab.com. https://blog.quarkslab.com/kubernetes-and-hostpath-a-love-hate-relationship.html. [Accessed 14-Jul-2023].

[21] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 122–137. https://doi.org/10.1145/3472883.3486974

[22] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[23] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst. *Proceedings of the VLDB Endowment* 13, 12 (aug 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[24] W. Richard Stevens. 1998. *UNIX Network Programming: Networking APIs: Sockets and XTI; Volume 1* (hardcover ed.). Prentice Hall. 1009 pages. https://lead.to/amazon/com/?op=bt&la=en&cu=usd&key=013490012X

[25] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.* 40, 1 (2017), 38–49.

[26] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 59–72. https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor

[27] Product Evangelist Yifat Perry. [n. d.]. Kubernetes CSI: Basics of CSI Volumes and How to Build a CSI Driver — bluexp.netapp.com. https://bluexp.netapp.com/blog/cvo-blg-kubernetes-csi-basics-of-csi-volumes-and-how-to-build-a-csi-driver. [Accessed 14-Jul-2023].