Diciclo: Data-Intensive Container Isolation for Multitenant Clouds

Building user-level services for multitenant container storage

USENIX ;login: (December 27, 2021)

Giorgos Kappes and Stergios V. Anastasiadis
Department of Computer Science and Engineering
University of Ioannina, Ioannina, Greece

Containers can suffer from poor performance because they share kernel resources for I/O-intensive workloads. We have developed user-level services that avoid system calls for POSIX-style filesystem storage and networking. In this article we explain the problems in current container systems, and how our Diciclo framework allows a key-value storage engine run faster over a distributed filesystem.

Containers are popular in the cloud landscape because they offer flexible virtualization with low overhead. Although their benefits follow from the native resource management of the operating system, the improved isolation of the container execution often requires hardware-level virtualization or rule-based control. In particular, the persistent storage of container images and data through the system kernel introduces several isolation and efficiency issues. Examples include the sub-optimal performance, security and fault-containment, or the excessive use of processing, memory and storage resources.

The data-intensive applications depend on scalable storage systems to handle their I/O requests. Along the I/O path, the control and data operations face contention at multiple layers of the shared network and storage infrastructure. A process invokes the application host kernel to access the storage servers backing the network filesystem or block volume. The host kernel handles the I/O calls through the local page cache and storage software stack. The storage servers serve multiple clients with the shared page cache, kernel modules, and storage device bandwidth of their local filesystems. The application hosts and storage servers compete for the available bandwidth and buffer space of the network cards and datacenter switch ports.

The network performance isolation has been previously addressed by centrally controlled resource allocators, flow endpoint rate limiters and congestion signaling protocols [7]. Promising techniques have also been introduced for the multitenant storage isolation. Although the dynamic resource allocation, kernel service replication and hardware virtualization are actively explored as potential solutions, the multitenant I/O handling through the operating-system kernel remains challenging.

First, the kernel services consume resources (e.g., memory space, cpu time) that are inaccurately accounted to the colocated processes due to the complex multiplexing or sharing. Second, the kernel consumes resources that often exceed the reservations of the tenants that cause the respective I/O activities (e.g., cpu time of page flushing). Third, there are consumable resources (e.g., software locks) that remain unaccounted for due to their allocation complexity (e.g., arbitration of synchronization instructions). Fourth, the kernel involvement incurs implicit hardware costs (e.g., mode switch, cache pollution, TLB flushes) that penalize unfairly the collocated tenants regardless of the relative intensity of their I/O activity.

Additional inefficiencies arise in cloned containers running on the same host. Indeed, their concurrent execution introduces duplication in the utilized memory and storage space, or the memory and I/O bandwidth. The multi-layer union filesystems and block-based copy-on-write snapshots reduce the waste of storage space. Yet, they do not always prevent the duplication of memory space and memory or I/O bandwidth at the host. Therefore, the resource contention and the inflexible sharing of the system kernel reduces the container I/O isolation and increases the resource duplication.

Current systems explicitly reserve resources for user-level execution, but they are blurry in the allocation and fair access of the kernel resources. Given the undesirable variability consequences, we argue that the effective container isolation requires two types of system support: *first, the explicit allocation of the hardware resources utilized at both the user and kernel level; and second, the fair access to the kernel operations and data structures.*

In Diciclo, we target the end-to-end performance isolation of the containerized data-intensive applications running on a shared infrastructure by different tenants. We enable the dynamic provisioning of scalable storage systems per tenant through the Polytropon toolkit [2]. Polytropon includes the libservice abstraction as a stackable user-level storage component derived from existing I/O libraries [3]. Based on the components of Polytropon, we build complex filesystem services that implement the storage client of a host or the

local filesystems of the storage servers. *We apply the same design pattern at either the client or the server to connect the libservices of a tenant to the application or server container processes through shared-memory interprocess communication.*

Containers and existing storage options

Containers are a lightweight virtualization abstraction provided by the operating system of a host to isolate a group of processes. They are typically supported by a userspace runtime, and kernel control mechanisms for resource allocation and isolation [6].

An *image* is a read-only set of application binaries and system packages organized as a stacked series of file archives (*layers*). It is made available from a registry running on an independent machine. A new container is typically created on the local storage of the host. The root filesystem is prepared by copying an image and expanding it into a file tree under a private directory. An additional writable layer is added over the expanded image to enable file modifications by the container execution. A storage driver of the container runtime allows sharing image layers across different containers through a union filesystem or snapshot storage.

A union filesystem offers a single logical view from multiple stacked directories (*branches*) supporting copy-on-write at the file level. The upper branch can be writable, while the remaining branches are read-only and shareable. The branches are stored on a local filesystem, or a distributed filesystem for scalability. Snapshot storage provides block-level copy-on-write. It is implemented by either a local filesystem, a local block device manager, or a remote block volume on network-attached storage.

For the storage of application data, a container can mount additional filesystems from the host (e.g., bind mount) or the network (e.g., volume plugin). The host provides persistence over a local or distributed filesystem. Alternatively, a volume plugin can mount from network storage either a filesystem or a block volume formatted with a local filesystem (e.g., Amazon EFS, EBS). The volume plugin is executed by the container runtime *with the necessary support from the kernel* (e.g., NFS kernel module). Finally, a container application may access cloud object storage (e.g., Amazon S3) through a RESTful API.

Applications access a filesystem with a system call that involves I/O directly (e.g., open) or indirectly (e.g., exec). The I/O requests are served inside the kernel or redirected from the kernel to a user-level process (e.g., FUSE [8]). Another option is to link an application to a library and implement the I/O functionality at user level, either as part of the application or at a different process over IPC.

Functional Requirements

The application hosts and storage servers require different types of data, filesystems, sharing, caching or deduplication [3].

Container Storage Systems A storage system consists of clients and servers, dynamically provisioned (e.g., root, application) or permanently operated (e.g., registry) by the provider. They serve the (i) image registry, (ii) the root filesystems that boot the containers, or (iii) the application filesystems with the application data. The image registry stores the container images of the root or application filesystem servers, and the applications. The root filesystem servers of a tenant are launched by having their images copied from the registry. The applications and filesystem servers are launched through clients accessing their images from the root filesystem servers. The application data is accessed through clients from the application filesystem servers.

File-based vs Block-based Storage A network storage client accesses the binaries or data in the form of blocks, files, or objects. We focus on block-based or file-based clients, which can efficiently serve the container storage needs. A *block-based client* serves the block volume on which we run a local filesystem. Treating entire volumes as regular files facilitates common management operations, such as migration, cloning and snapshots, at the backend storage. Despite this convenience, a block volume is only accessible by a single host and incurs the overhead of mounting the volume and running on it a local filesystem. In contrast, a *file-based client* natively accesses the files from a distributed filesystem. Multiple hosts directly share files through distinct clients, but with the potential server inconvenience of managing application files rather than volumes. The file-based or block-based clients are widely used in container storage and we should support them both.

Caching and Deduplication The root filesystems of a tenant should be isolated and managed efficiently. The efficiency refers to the *storage space* of the backend servers, the *memory space* of the container hosts, and the *memory* or *network bandwidth*.

The *block-based storage* accommodates a root filesystem over a separate block volume possibly derived from an image template. The backend storage supports volume snapshots to efficiently store the same blocks of different volumes only once. Under the volume clients, a shared cache transfers once the common volume blocks of the tenant over the network. A separate cache in the root filesystem lets the container reuse the recently accessed blocks without additional traffic to the shared client cache.

Alternatively, the *file-based storage* accommodates a root filesystem over a shared network filesystem accessed by the client of the tenant. A separate union filesystem over each root filesystem deduplicates the files of different container clones to store them once at the backend. We transfer once to

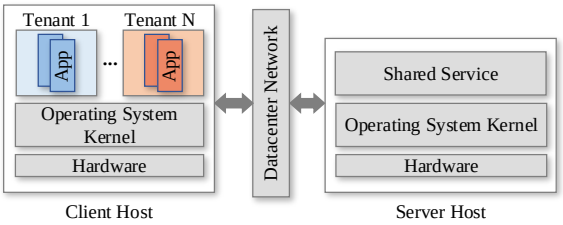

Figure 1: Typical approach for serving multiple tenants in existing cloud infrastructures.

the host the common files of the root filesystems with a shared cache inside the distributed filesystem client. The caching and deduplication procedures described above can be similarly applied to application filesystems.

Overview

We target the data-intensive applications running in a multi-tenant cloud infrastructure. Both the applications and their storage servers run in containers launched by an orchestration system over images retrieved from a scalable storage system. We introduce the Diciclo unified framework to provision per tenant container storage systems with the following five goals [3, 6]:

1. **Isolation** Improve the performance isolation and the fault containment of data-intensive tenants colocated on the same client machine.

2. **Compatibility** Provide native container access to scalable persistent storage through a backward-compatible POSIX-like interface (e.g., open, fork, exec).

3. **Efficiency** Serve the root and application filesystems of containers through the effective utilization of the processor, memory and storage resources.

4. **Elasticity** Permit the dynamic resource allocation per tenant according to the configured reservations and utilization measurements.

5. **Flexibility** Enable flexible tenant configuration of the sharing and caching policies (e.g., files, consistency).

Our key idea is to move parts of the system kernel to user level and provide separate instances of critical services to each tenant on every machine. Accordingly, we introduce a unified architecture that allows each tenant to run both the applications and I/O services at user level on per-tenant reserved resources (e.g., cores, memory, ports) and avoid contention with colocated entities. The cloud applications typically run at client machines that access server machines over the datacenter network. In existing infrastructures, the applications run inside containers over the shared operating system of

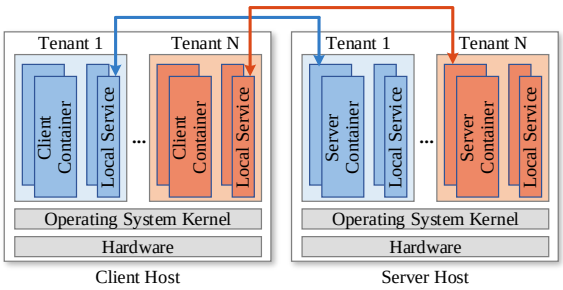

Figure 2: Diciclo uses distinct user-level services per tenant at both the client and server side of storage.

each client machine. Correspondingly, the server machines run common services (e.g., distributed storage) that are shared across the different tenants (Fig. 1).

In the Diciclo architecture, we recognize that both the applications and their remote services can fully run at user level rather than relying heavily on the local system kernel (Fig. 2). Essentially, both the client and server machines run the application and the server processes in containers on resources separately reserved for each tenant. The application and server containers run at user level and access over user-level interprocess communication their local services. The local services themselves run at user level and provide access to local devices, such as storage, network and accelerator cards. Thus, we devote to each tenant its own application and server containers along with their interprocess-communication and local services.

Diciclo achieves several benefits. (i) We offer more predictable behavior because each tenant depends on private resources and software components. (ii) We achieve resilience because we minimize the reliance on the common attack surface of the operating-system kernel. (iii) We provide elasticity because we can dynamically deploy the software infrastructure for each tenant. This includes not only the applications, but also the client and server side of distributed storage systems and other facilities (e.g., storage engine, local filesystem, network stack software).

Design

The *container pool* is a collection of containers and namespaces on a machine from a tenant. The *filesystem service* is a collection of user-level I/O services implementing a local or network filesystem (Fig. 3). The *libservice* is a user-level layer implementing a particular filesystem component. The *container engine* is responsible to initiate the container pools and allocate resources elastically according to the recorded reservations and utilizations. A process obtains access to the filesystem service through a preloaded or linked *filesystem library* and communicates with the filesystem service over shared memory.

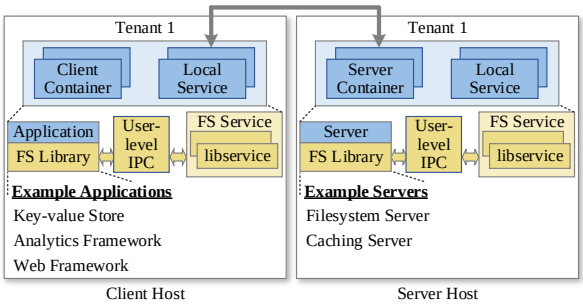

Figure 3: Diciclo provides a common framework for the implementation of both the client and server services.

In order to achieve our goals, our design is based on the following four principles [6]:

1. **Dual interface** Support common application I/O calls through the filesystem library, and only use the kernel for legacy software or system calls with implicit I/O.

2. **Filesystem integration** The libservices of a filesystem instance interact directly with each other through function calls for speed and efficiency.

3. **User-level execution** The filesystem service and the default communication path both run at user level for isolation, flexibility and reduced kernel overhead.

4. **Path isolation** A filesystem service isolates the storage I/O of a container pool at the client side in order to reduce the interference among the colocated pools of the same or different tenants and improve their flexibility.

Libservices The libservice is a standalone component of a filesystem service [3]. It provides a storage function, such as caching, deduplication, key-value store, local filesystem, network access to a file or block storage system. A libservice is derived from an existing library that runs a storage function at user level and makes it accessible through a POSIX-like interface (e.g., FUSE, LKL, Rump).

An existing library cannot be used out-of-the-box as libservice for several reasons. First, if a library is linked or preloaded directly to a process, it complicates the multiprocess sharing. Second, the I/O routing through the kernel (e.g., FUSE) introduces overheads from frequent mode switches and memory copies. The cost is even higher if more than one libraries are combined to a complex service. Third, the I/O routing through the kernel causes the shared structure contention that we strive to avoid. We overcome the above limitations by separating the filesystem libraries from their framework interface (e.g., FUSE). We port the library to an independent libservice with POSIX-like I/O functions expanded to include the libservice object as parameter without dependencies from global variables. A filesystem service is constructed from a list of stacked libservices, or a tree of libservices more generally.

Client The client side serves the data-processing applications, the container engines that instantiate the application or server containers of the tenants, and the orchestration system that loads the container images to the root filesystem servers. The filesystem service of an application filesystem provides either (i) a shared cache with the client of a network filesystem, or (ii) a shared local filesystem with cache on a network block volume. On the other hand, the filesystem service of a container root filesystem provides either (i) a union filesystem on a shared network filesystem client with cache, or (ii) a local filesystem with cache over a network block volume with shared block cache.

Server The filesystem service of a storage server maintains data and metadata chunks on the local storage devices. Depending on their I/O characteristics (e.g., size, rate), the chunks can be stored over a single local filesystem, or a local filesystem and a key-value store. Each local filesystem or key-value store can have a separate cache. For fast recovery from a crash, the received chunk updates can be appended to the write-ahead log of the key-value store and the journal of the local filesystem, or a write-ahead log over a common local filesystem used for both the data and metadata. We can use a separate libservice to implement each of the local filesystem, key-value store, write-ahead log or journal. The server processes communicate over shared memory with the filesystem service and over the network with the clients. The server processes and filesystem services run isolated in the container pool of the tenant on the storage server machines.

Interprocess Communication (IPC) A filesystem service is invoked by an application or server process through user-level IPC to minimize mode switches and processor cache stalls. The dual interface that we support, alternatively allows an application to invoke regular system calls for increased legacy compatibility. The user-level IPC component is implemented with circular queues over shared memory inside the IPC namespace of the container pool. We relax the ordering semantics of the queue items in order to combine high concurrency with low latency [4]. The I/O requests of the application and server processes are placed in the circular queue to be extracted by the filesystem service. The request passes through the top libservice of the filesystem service before it will reach the local storage device or the network client if necessary. The response is sent back without queue involvement through a shared-memory buffer prepared by the sender and referenced by the request. We have developed the Asterope optimization methodology to achieve fast memory copy through the specific architectural characteristics of each system platform [5].

Local Resource Management The resource and device management at each host critically isolate the container pools running the applications, servers, and their libservices. The resource reservation approximately guarantees a specified

amount of cpu shares, memory space, local storage space, and I/O bandwidth of local network or storage devices. The resource management tracks the resources, accounts them to processes and dynamically allocates them according to the current reservation and utilization. The device management provides protected operation of the local devices among the processes of the colocated tenants.

For the resource and device management we rely on kernel mechanisms, such as the Linux cgroups (v1/v2) and device drivers. This decision is justified by the satisfactory accuracy of their hardware resource accounting to user-level processes. Based on the above kernel mechanisms, we build the filesystem services and their interprocess communication running at user level. Alternatively, it is possible to also manage the network cards and storage devices directly from user level to achieve high-performance access.

Example Storage Systems

Below we describe examples of container storage systems that can be built from libservices. The clients and servers are initiated by the container engines or the orchestration system.

Container image storage system is a registry of container images maintained by the provider or a tenant. The server stores image archive files over libservices for persistent local storage with cache and block-level deduplication (e.g., ZFS). The client uses a libservice for network storage access with cache (e.g., NFS). The system distributes the container images of the root filesystem servers, the application filesystem servers, or the applications.

Root filesystem storage system is dynamically provisioned to boot the applications and application filesystem servers. The server provides persistent local storage with cache and block-level deduplication. It is based on libservices of a local journaled filesystem (e.g., ext4) or a recoverable key-value store (e.g., RocksDB). The client supports efficient container cloning based on libservices of a union filesystem (e.g., OverlayFS) over a shared network storage client and cache (e.g., CephFS). Alternatively, it is built from the libservices of the journaled filesystem with cache (e.g., ext4) on a network block device (e.g., RBD) and a shared cache (e.g., Redis). A container engine starts a client in a pool to mount the root filesystems of the containers running the applications or the application filesystem servers.

Application filesystem storage system is dynamically provisioned to serve the data-intensive or stateful applications. The server provides persistent local storage with cache based on the libservice of a local journaled filesystem (e.g., ext4) or a recoverable key-value store (e.g., RocksDB). The client libservice provides network storage access with cache (e.g., CephFS).

Danaus Prototype The Danaus client [6] consists of the *filesystem library* linked to each application, the *filesystem services* that handle the storage I/O, and the interprocess communication that connects the applications with the services (Fig. 4). The *front driver* of the filesystem library passes the incoming requests at user level to the *back driver* of a filesystem service over shared memory. A *filesystem service* is a standalone user-level process that runs the *filesystem instances* mounted to the containers. A filesystem instance is implemented as a stack of libservices. Our prototype implementation currently supports the *union libservice* of the union filesystem and the *client libservice* of the backend client. The union libservice invokes directly the client libservice to serve the branch requests without extra context switching or data copying.

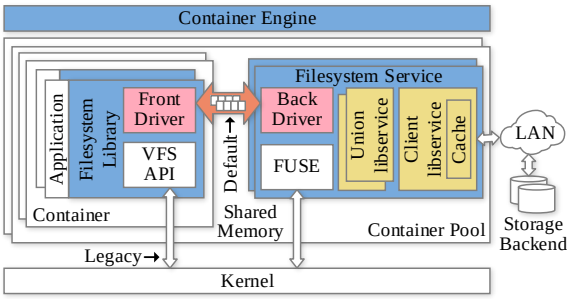

Figure 4: Implementation of the Danaus user-level filesystem client based on Diciclo.

Performance Evaluation

Our testbed consists of two 64-core AMD-based servers running Debian Linux (kernel v4.9) [6]. The one hosts the application containers and the other uses 7 Xen VMs with ramdisk devices to run a Ceph [9] storage cluster. We run the RocksDB storage engine independently in 32 container pools of the same machine. Each pool is configured with one container over 2 cores and 8GB RAM. The container runs RocksDB with 64MB memory buffer and 2 compaction threads. The pool uses a dedicated Ceph client to mount a private root filesystem holding the files of both the container and RocksDB. We measured the average put latency of RocksDB for 1-32 pools over clients based on Danaus (D), FUSE (F) or kernel (K) [6]. In Fig. 5a, D is faster than F and K up to 5.9x and 16.2x (32 pools). In another experiment, we run an out-of-core read-intensive workload per pool. In Fig. 5b, D is faster than F and K up to 1.4x and 2.2x (32 pools).

With extensive kernel profiling, we found that the kernel path limits the tenant I/O isolation for two reasons. First, the pools compete in the kernel on shared data structures (e.g., filesystem metadata) causing excessive lock wait time. Second, the I/O handling leads to kernel background activity (e.g., dirty page flushing) on resources (e.g., cores) of pools unrelated to the activity.

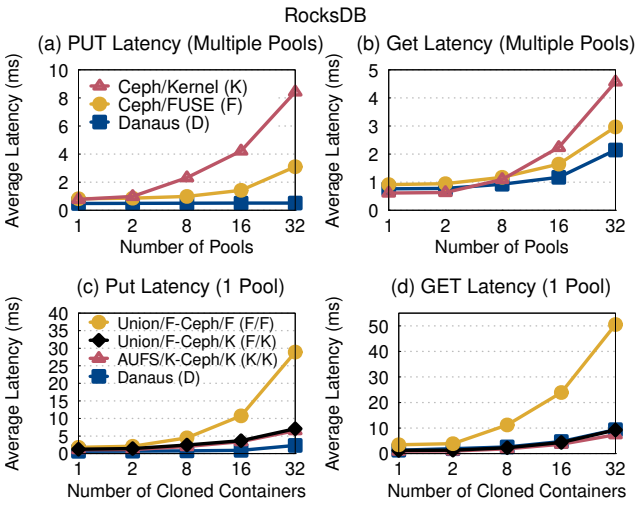

Figure 5: Average latency of RocksDB put and get in scaleout (a,b) and scaleup (c,d) settings of container pools.

We also explored a scaleup setting running multiple cloned containers in a single pool. Each container runs a private RocksDB instance. A container mounts its root filesystem through a private filesystem instance consisting of a distinct union filesystem and a *shared* Ceph client. The Ceph client lets the containers share the lower read-only layer of their root filesystem. We measured the RocksDB latency on Danaus or different combinations of a union filesystem over a Ceph client, each implemented in FUSE or inside the kernel (F/F, F/K, K/K). With respect to put latency, D is faster than F/F, F/K and K/K up to 12.6x, 3.9x and 3.6x, respectively (Fig. 5c). In contrast, the get workload (Fig. 5d) leads to mixed results with D up to 5.4x faster than F/F at 32 clones but up to 2x slower than K/K at 2 clones.

We conclude that Danaus reduces the kernel lock contention and resource consumption by running and accessing *at user level* either a distinct client per container, or a shared client across the cloned containers.

Related Work and Discussion

A library operating system (libOS) reduces the isolation dependence from the kernel resource management because it shifts functionality from the kernel to the user level (e.g., Demikernel [10]). Yet, the typical linking of a libOS to a single process complicates the multiprocess state sharing of typical applications. In a similar sense, a filesystem provides flexible resource management when it is accessed through the kernel but runs at user level. However, it sacrifices performance and efficiency due to the extra data copying and mode or context switching (e.g., FUSE [8]). The container isolation and resource efficiency can be addressed with improved scalability in the state management of the system kernel [1]. Nevertheless, refactoring the kernel for scalability is a long-standing challenging problem.

As a pragmatic alternative, we relocate to user level both the system functionality and the interprocess communication. Our approach can be beneficial for a broad range of system components, including the local and remote access to different types of data storage, or the network communication.

Lessons Learned

We learned several lessons from our experience with the development of Diciclo: (i) The system kernel can become hotspot for the colocated containers. (ii) The user-level execution and interprocess communication of filesystems improve performance and isolation. (iii) The user-level implementation of a filesystem service facilitates the code development because it reduces the kernel dependencies. (iv) The user-level execution of I/O services improves the latency and throughput stability under contention conditions. (v) Danaus can provide an integrated approach to serve both the root images and application data of containers. (vi) In scaleout workloads, the colocated tenants can be served by distinct user-level filesystems for improved scalability, while in scaleup workloads avoiding completely the kernel also prevents several overheads.

References

[1] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Transactions on Computer Systems*, 32(4), January 2015.

[2] Giorgos Kappes and Stergios V. Anastasiadis. A User-Level Toolkit for Storage I/O Isolation on Multitenant Hosts. In *ACM Symposium on Cloud Computing*, pages 74–89, Virtual Event, USA, 2020.

[3] Giorgos Kappes and Stergios V. Anastasiadis. Libservices: Dynamic Storage Provisioning for Multitenant I/O Isolation. In *ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 33–41, Tsukuba, Japan, August 2020.

[4] Giorgos Kappes and Stergios V. Anastasiadis. A Lock-free Relaxed Concurrent Queue for Fast Work Distribution. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 454–456, Virtual Event, Republic of Korea, February 2021.

[5] Giorgos Kappes and Stergios V. Anastasiadis. Asterope: A Cross-Platform Optimization Method for Fast Memory Copy. In *Workshop on Programming Languages and Operating Systems (in conjunction with ACM Symposium on Operating Systems Principles)*, pages 9–16, Virtual Event, Germany, October 2021.

[6] Giorgos Kappes and Stergios V. Anastasiadis. Experience Paper: Danaus: Isolation and Efficiency of Container I/O at the Client Side of Network Storage. In *ACM/IFIP International Middleware Conference*, pages 132–145, Virtual Event, Canada, December 2021.

[7] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *ACM Symposium on Operating Systems Principles*, pages 182–196, Farminton, Pennsylvania, 2013.

[8] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *USENIX Conference on File and Storage Technologies*, pages 59–72, Santa Clara, CA, February 2017.

[9] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, Seattle, WA, November 2006.

[10] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *ACM Symposium on Operating Systems Principles*, pages 195–211, Virtual Event, Germany, October 2021.

Giorgos Kappes is Postdoctoral Researcher in the Department of Computer Science and Engineering at the University of Ioannina in Greece. His research activities include the design, implementation, and evaluation of user-level services for multitenant cloud environments. His research interests include operating systems, data storage, and security. He received BSc, MSc, and PhD degrees in Computer Science from the University of Ioannina, Greece. Email: gkappes@cse.uoi.gr

Stergios V. Anastasiadis is Director of the Computer Systems Lab and Associate Professor at the Department of Computer Science and Engineering, University of Ioannina (Greece). In the past, he held visiting positions at University of Cambridge (UK), EPFL (Switzerland), Duke University (NC, USA) and HP Labs (CA, USA). His research interests include operating systems, data storage, security and cloud computing. He received MSc and PhD degrees in Computer Science from the University of Toronto (Canada). Email: stergios@cse.uoi.gr