Fisc: A Large-scale Cloud-native-oriented File System

Qiang Li, *Alibaba Group;* Lulu Chen, *Fudan University and Alibaba Group;* Xiaoliang Wang, *Nanjing University;* Shuo Huang, *Alibaba Group;* Qiao Xiang, *Xiamen University;* Yuanyuan Dong, Wenhui Yao, Minfei Huang, Puyuan Yang, Shanyang Liu, Zhaosheng Zhu, Huayong Wang, Haonan Qiu, Derui Liu, Shaozong Liu, Yujie Zhou, Yaohui Wu, Zhiwu Wu, Shang Gao, Chao Han, Zicheng Luo, Yuchao Shao, Gexiao Tian, Zhongjie Wu, Zheng Cao, and Jinbo Wu, *Alibaba Group;* Jiwu Shu, *Xiamen University;* Jie Wu, *Fudan University;* Jiesheng Wu, *Alibaba Group*

https://www.usenix.org/conference/fast23/presentation/li-qiang-fisc

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by

Fisc: A Large-scale Cloud-native-oriented File System

Qiang Li[◇], Lulu Chen[†◇], Xiaoliang Wang[‡], Shuo Huang[◇], Qiao Xiang[⋆], Yuanyuan Dong[◇], Wenhui Yao[◇], Minfei Huang[◇], Puyuan Yang[◇], Shanyang Liu[◇], Zhaosheng Zhu[◇], Huayong Wang[◇], Haonan Qiu[◇], Derui Liu[◇], Shaozong Liu[◇], Yujie Zhou[◇], Yaohui Wu[◇], Zhiwu Wu[◇], Shang Gao[◇], Chao Han[◇], Zicheng Luo[◇], Yuchao Shao[◇], Gexiao Tian[◇], Zhongjie Wu[◇], Zheng Cao[◇], Jinbo Wu[◇], Jiwu Shu[⋆], Jie Wu[†], Jiesheng Wu[◇], [◇]Alibaba Group, [†]Fudan University, [‡]Nanjing University, [⋆]Xiamen University

Abstract

Despite the progress of cloud-native technologies, existing distributed file systems are ill-suited for multi-tenant cloud-native applications for two reasons, First, their clients are typically heavyweight, resulting in a low level of resource multiplexing among containers. Second, their architecture is based on network gateway and falls short in providing efficient, highly-available and scalable I/O services for cloud-native applications. In this paper, we propose Fisc, a large-scale, cloud-native-oriented distributed file system. Fisc introduces three key designs: (1) a lightweight file system client to improve the multiplexing of resources with a two-layer resource aggregation, (2) a storage-aware distributed gateway to improve the performance, availability and scalability of I/O services, and (3) a DPU-based virtio-Fisc device offloading key functions to hardware. Fisc has been deployed in production for over three years and now serves cloud applications running over 3 million cores. Results show that Fisc client only consumes 69% CPU resources compared to the traditional file system client. The production environment shows that the average latency of online searching tasks is less than 500 μs when they access their files in Fisc.

1 Introduction

Many applications, such as data analytics [1], machine learning [2], and transactional workflows [3, 4] are deployed in public clouds. The emerging cloud-native technologies are shifting virtualization in clouds from virtual machines (VM) to containers and pushing up the abstraction provided to tenants from resources (*e.g.*, CPU and memory) to services (*e.g.*, database and object storage service). As such, cloud service providers (CSPs) must rethink their fundamental services to provide efficient, flexible support to cloud-native applications.

Specifically, file system (FS) is one such fundamental service, with which applications can store and access their data [5–7]. Tenants typically employ FS in the cloud in one of two modes. They either purchase cloud storage (*e.g.*, SSD) and deploy their own FS, or directly use the FS service provided by CSPs. As CSPs gradually switch from server- centric to resource-disaggregated architectures, tenants increasingly use the second approach for its elasticity, flexibility, on-demand charging, and ease of use [6, 8, 9].

File systems need to be redesigned to support cloud-native applications. Existing distributed file systems (*e.g.*, [5–7]) are ill-suited for multi-tenant cloud-native applications for two reasons. First, clients in these systems have a low level of resource multiplexing among containers. That hinders CSPs from achieving high efficiency of resources and makes it difficult for each computation server to support a large number of containers for cloud-native applications. Specifically, these clients typically adopt a heavyweight design to provide many functionalities, including interfaces for interacting with applications, storage protocols for data persistence and failure handling, network-related functions for communications with data nodes and metadata masters, and security-related functions for authorization checking. As such, each client needs to reserve many exclusive resources, and a server can host only a small number of containers concurrently, resulting in inefficient use of resources.

Second, a centralized network gateway employed for file system service in the cloud cannot satisfy the requirement of cloud-native applications for performance, availability, and load balancing. A network gateway is a component that connects clients in the virtual domain of users to backend proxies in the physical domain of CSPs. This network-gateway-based architecture has a series of limitations, including (1) a sub-optimal, ms-level latency to pass through the gateway, (2) the incapability of data locality optimization and fast failure handling due to the unawareness of file semantics and storage protocols, (3) the incompatibility with high-performance network stack like RDMA without intrusive changes to clients, and (4) the load balancing gap between network connections and files. Besides, to match the throughput of a large-scale file system of thousands of nodes, it would take non-negligible costs for CSPs. Luna and Solar [10] propose storage network stacks for Alibaba's EBS service. However, they only focus on achieving high performance within the physical domain of CSPs. They cannot provide high performance for the whole

path from the file clients in the virtual domain of users to the storage clusters in the physical domain of CSPs.

Fisc: a cloud-native-oriented file system. In this paper, we design Fisc, a cloud-native-oriented distributed file system service to provide cloud-native applications with high-performance, high-availability storage services at low cost. Fisc consists of two key components: lightweight clients and a storage-aware distributed gateway (SaDGW).

First, with a two-layer aggregation, Fisc moves user-unaware functionalities (*e.g.*, network stacks and storage protocols) out of clients in the containers and offloads them to the Data Processing Units (DPU) of computation servers and the backend storage nodes of CSPs to aggregate their resources, respectively. As a result, the resources used for these functionalities can be fully multiplexed, lowering the amortized cost. Meanwhile, as each client consumes substantially fewer resources, a computation server can host a large number of containers for cloud-native applications.

Second, Fisc introduces SaDGW to provide a direct highway with a high-performance network stack [10] between the computation and storage servers. Specifically, we leverage the file system semantics on the highway path to build a storage-aware routing mechanism to route clients' file requests from the frontend virtual domain of tenants to the backend physical domain of CSPs with a granularity of files instead of network flows. We design a series of mechanisms, such as storage-aware failure handling and locality-aware read optimization, to improve the availability of Fisc. We have also employed a file-based fine-grained scheduling mechanism to balance loads of proxies at storage nodes.

Implementing Fisc with a software-hardware co-design. Realizing lightweight clients and SaDGW completely in software is inefficient. As such, we leverage the emerging DPUs to implement part of the functionalities of clients and the core functionalities of SaDGW. We adopt a virtio-Fisc device in DPU to offload the network stacks and storage protocols and provide secure and high-efficient passthrough from the users' virtual domain containers to the file system of CSP's physical domain. We also leverage the fast path in DPU to accelerate the I/O processing, further improving the performance of Fisc.

Production deployment. Fisc has been deployed in production DCN for three years and serves applications running on over 3 million cores in Alibaba. For large-scale development, it presents an abstracted virtual RPC (vRPC) based on SaDGW and virtio-Fisc devices, which is easy to use and can be adopted by other cloud-native services like Function as a Service (FaaS). Compared to the on-premise Pangu client, the CPU and memory consumption of the Fisc client is reduced by 69% and 20%, respectively. The availability is improved by an order of magnitude (*e.g.*, failure recovery from a second-level to a 100ms-level). For the online-search query service, its average and P999 latency in Fisc are <500 μs and <60 ms, respectively. Its average latency jitter is less than 5%.

2 Background and Motivation

2.1 File Systems

File system (FS) is a fundamental service for users to store and access their data. Large-scale distributed file systems like Tectonic [7], Colossus [5], and Pangu [6] have been developed by different companies in their datacenters. Generally, they consist of three components, *masters, data servers,* and *clients*. The *masters* manage data servers and maintain the metadata of the whole system (*e.g.*, the file namespace and the mapping from file chunks to data servers). The *data servers* are storage nodes responsible for managing file chunks and storing their data on storage media (*e.g.*, HDDs and SSD). The *clients* interact with the masters for metadata and the data servers for data. Notice that clients in representative large-scale file systems (*e.g.*, Tectonic [7] and Colossus [5]) are heavyweight. They provide complex functions, including not only storage protocols for data persistence and failure handling but communication with masters and data servers, as well as security-related functions such as authorization.

Pangu [11] is a large-scale distributed storage system in Alibaba and provides append-only file semantics like HDFS [12]. It works as a unified storage core of Alibaba Cloud. Multiple businesses (*e.g.*, Elastic Block Service [10, 13], Object Storage Service [14], Network Attached Storage [15], and MaxCompute [16]) are built on top of Pangu. They adopt the Pangu clients for persistent, append-only file storage, employ a key/value-like index mapping to update data, and use a garbage collection mechanism to compress historical data.

2.2 Cloud Native

With the development of cloud-native technology (*e.g.*, microservice, container, and serverless computing), more and more tenants are deploying their applications into the public cloud and directly using the services provided by CSPs (*e.g.*, database and object storage service). In 2020, Alibaba also migrated all its core businesses, such as Taobao and Tmall, to cloud-native containers. Cloud-native technologies substantially simplify the development and operation of tenants and demonstrate two characteristics. First, with fine-grained containers being used instead of VMs, the number of containers in a computation server can exceed 1000 [17, 18], *i.e.*, $\sim$10 times more than that of VMs. Second, cloud-native technologies push up the abstraction provided to tenants from VMs to services. The implementation of services is transparent to tenants but must provide high performance under heterogeneous workloads. To this end, bare-metal DPUs are increasingly used to accelerate cloud-native applications. For example, AWS adopts Nitro [19] and Alibaba adopts X-Dragon [20, 21]. These bare-metal DPUs utilize the virtio technology for I/O virtualization and can provide high-performance support to a broad range of cloud services.

2.3 Motivation

Cloud-native applications bring new challenges for CSPs to provide file system service.

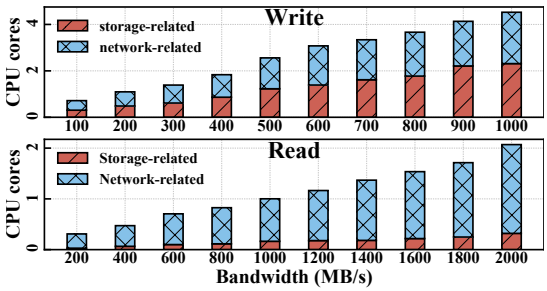


Figure 1: The CPU consumption of an HDFS client under different I/O bandwidths.

Isolated file system clients cause low resource utilization. In traditional file systems [5, 7], a client is responsible for multiple tasks, including storage protocols of reliability and consistency, failure handling, network-related functions, and authorization-related functions. As such, applications usually pre-allocate I/O threads and reserve memory resources and network connections for file system clients. Because the resources of FS clients in containers are isolated from each other, the resource utilization of CSPs is low. As a result, achieving a high density of over 1,000 containers in a computation server is difficult. Take the resource consumption of an HDFS client in a Hadoop-2.10.2 cluster of three Intel(R) Xeon(R) Gold 5218 servers as an example. Figure 1 plots the CPU consumption of the HDFS client under different read and write bandwidths. Even if the client writes files at a bandwidth of 200 MB/s, it consumes 1.1 CPU cores. Consider a typical scenario where a container is allocated two cores. It means over 50% CPU resources are spent on I/O.

We make a key observation that many common functions (*e.g.*, storage protocols and network stacks) of different FS clients can be aggregated to achieve more efficient resource sharing. With this aggregation, we can provide a lightweight file system interface for different tenants, and simplify the maintenance and upgrade of FS clients.

Network gateway becomes the bottleneck. FS clients of cloud-native applications are in the virtual domain of users, while the file system resides in the physical domain of CSPs. For security reasons, clients cannot directly access the file system but have to use a network gateway (*i.e.*, network load balancer) to access the data. However, this network gateway cannot satisfy the requirements of cloud-native applications on file services in terms of performance, availability, load balance, and cost.

- *Performance.* Performance-critical cloud-native applications (*e.g.*, interactive applications [22]) require a 100μs-level storage access latency. Although file systems such as Pangu are equipped with high-performance SSD and RDMA in the backend cluster [6], which provides a 100μs-level latency, an I/O request needs to go through multiple hops in a network-gateway-based architecture, resulting in a second-level or ms-level latency [23, 24].
- *Availability.* Cloud-native applications often require a ms-level recovery latency [25] in the case of storage system failures (*e.g.*, network jitters and server breaking down). However, with a network gateway, file systems can only support second-level failure handling [26, 27] due to the gap between files and network connections. Specifically, the network-connection-based Service Level Agreement (SLA) is substantially different from the file-based SLA of file systems. As such, it is hard to leverage storage protocols in a network-gateway-based architecture to improve the availability of file systems.
- *Load balance.* The network gateway distributes the load to different proxies based on the number of network connections. That may lead to a significant load imbalance of files among the proxies due to the semantics gap between files and connections. For example, the load among proxies can be as much as ten-fold different in the NAS service in Alibaba Cloud [15]. In addition, the gateway may direct a read request to a storage server with no requested data. The server must forward the request to another storage server that has the data, which will amplify the traffic.
- *Cost.* A large-scale file system requires a large amount of hardware dedicated to the network gateway in order to match the total throughput of its storage cluster, which typically consists of thousands of storage nodes. Given a cluster of 10,000 storage nodes, each of which is equipped with a 25$\times$2 Gbps NIC, its total throughput is 500 Tbps. If the throughput of a network gateway machine is 100 Gbps, we need 5,000 gateway machines to match the total throughput of the cluster, which introduces a non-negligible cost for CSPs.

3 Overview of Fisc

In this section, we give an overview of Fisc, including its design rationale, architecture and basic workflow.

3.1 Design Rationale

Aggregating the resources of FS clients. Resource aggregation is the nature of cloud computing, which can improve resource utilization and provide elastic, efficient, and on-demand cloud service. In contrast to the traditional resource-intensive FS clients, we aggregate functions like storage protocol and network-related functions by offloading them to the CSP's domain (*e.g.*, the DPUs at computation and storage servers). Meanwhile, this aggregation allows CSP to provide a reservation-only interface with a lightweight client for cloud-native applications. As such, it allows a computation server to host a large number of application containers concurrently.

Storage-aware distributed gateway. Instead of using a centralized network gateway, we resort to a distributed storage-aware gateway to set up direct highways between each computation server and its corresponding remote storage nodes. This design allows us to adopt high-performance network protocols connecting the virtual and physical domains. It also leverages storage semantics on the highways to improve the availability and locality of file access requests and guarantee the load balance among storage nodes.

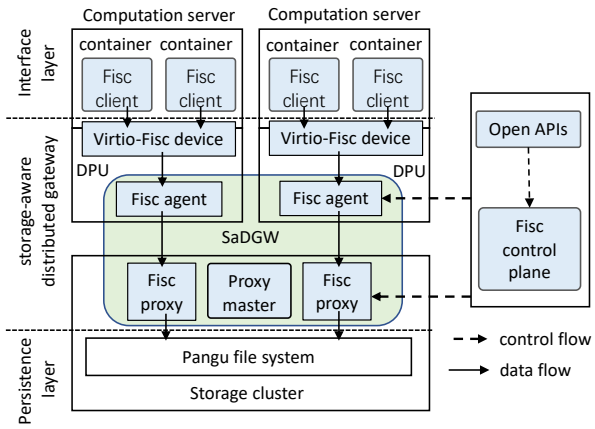


Figure 2: The architecture of Fisc.

Software and hardware co-design. To improve the efficiency and performance of the file system service, we leverage the emerging DPUs deployed in physical servers. Through careful hardware and software co-design, we can implement secure, efficient passthrough from the users' containers in the virtual domain to the file system of CSP's physical domain. Moreover, we can also introduce a fast path in DPU to accelerate the I/O processing.

3.2 Architecture

As shown in Figure 2, Fisc consists of a control plane and a data plane. The control plane provides open APIs for tenants to create Fisc FS instances, mount the Fisc FS to their VM/containers, and allocate virtio devices to accelerate the passthrough from the virtual domain to the physical domain.

Fisc's data plane consists of three layers: interface layer, storage-aware distributed gateway, and persistence layer. The lightweight Fisc client is placed in the frontend, which provides FS service interfaces for applications. The distributed storage-aware distributed gateway (SaDGW) is in the middle layer, composed of Fisc agents in the DPU of each computation server, Fisc proxies in each storage node, and a group of Fisc proxy masters in the storage cluster. The Fisc proxy masters are responsible for managing Fisc proxies and Fisc agents. The backend persistence layer is Pangu, which is responsible for processing the requests and persisting the data in storage media.

Lightweight Fisc client. The aggregation of client resources occurs at the Fisc agent in the DPU of each computation server and the Fisc proxy in each storage server. We dissect the functions of FS clients and make careful aggregation tradeoffs to decide where these functions should be aggregated (*i.e.*, Fisc agents or proxies). We also design mechanisms to simplify the implementation of Fisc clients and maintain compatibility across different versions of their software libraries.

SaDGW. This gateway gives full play to the 100μs-level high-speed SSD and RDMA technologies via direct and high-performance network connections between Fisc agents and Fisc proxies. Based on the file granularity routing in each Fisc Agent, it leverages the storage semantics on the route to eliminate the gap between network and file to achieve a P999 ms-level SLA. Moreover, it implements a locality-aware read mechanism that avoids the read traffic amplification and doubles the read throughput.

HW and SW co-design on DPU. Fisc provides a virtio-Fisc device to build up secure and efficient passthrough from virtual containers to the physical storage cluster. Based on the device, a co-designed FPGA cache is presented as a fast path to further improve Fisc's performance. With regard to the scarce resource of DPU, optimizations for CPU, memory, and network are proposed.

With these three modules, we further provide a vRPC (virtual RPC) abstraction for storage service, which can be easily adopted by cloud-native services. Besides, Fisc adopts an end-to-end (E2E) QoS mechanism for different priority applications like online search and offline training. With proxy master scheduling, Fisc builds up file-granularity load balancing among Fisc Proxies, which avoids the imbalance caused by traditional network connection-based scheduling.

3.3 Workflow of Fisc

In the control plane, when a tenant calls the open APIs to create a Fisc instance, Fisc control plane maps the instance to the backend Pangu file system, and pushes the information of the tenant and the mount point to the Fisc proxy masters deployed in the Pangu storage cluster. The Fisc proxy master pushes the proxy mapping (*i.e.*, the mapping between the mount point and the Fisc proxies) to the Fisc agent whenever the tenant attaches the mount point to a VM/container. In the end, the control plane attaches a virtio-Fisc device to the corresponding VM/container.

The workflow of the data plane mainly involves SaDGW with a fine-granularity route table. Given a meta operation request of files of the mount point, it arrives at the Fisc agent through the virtio-Fisc device. The Fisc agent randomly chooses a Fisc proxy according to the mapping between the mount point and Fisc proxies. If it is an open operation for a file, a route entry associated with the opened file will be constructed with its file handle and the Fisc proxy location. Afterwards, the subsequent read/write requests of the file will be routed according to the route entry. More details of storage-aware routing optimizations are in §4.2.

4 Design and Implementation

4.1 Lightweight Fisc Client

We adopt a lightweight design for Fisc clients by offloading most of their functions to Fisc agents in the DPU of computations servers and Fisc proxies on the storage nodes. Through this two-layer function aggregation, Fisc achieves a high level of resource multiplexing. In addition, we also introduce a unified RPC-based method to simplify the implementation of Fisc clients and a mechanism similar to Protocol Buffers (PB) to maintain compatibility across their different versions.

4.1.1 Function Offloading and Aggregation Tradeoff

Typical heavyweight FS clients [5–7] provide four types of functions: (1) file interfaces and structures (*e.g.*, APIs and file handlers), (2) storage-related protocols (*e.g.*, replication reliability, data consistency, and failure handling), (3) security and authentication (*e.g.*, authorization checking) and (4) network-related protocols (*e.g.*, RPC with data nodes or metadata nodes). We make a *key observation* that in cloud-native applications, users are only interested in the first type of functions and the implementations of other functions are transparent to users. Therefore, we can move the latter three functions out of Fisc clients and aggregate them to achieve a high level of multiplexing on resources. However, the locations where they are aggregated (*i.e.*, Fisc agents or Fisc proxy) have a great impact on the effects of multiplexing. We elaborate on our offloading designs of different functions.

Offloading network-related functions to Fisc agent. We offload the network-related functions of conventional clients to the Fisc agent in the DPU of the computation server. This is motivated by the recent success of DPU-based high-performance network stacks (*e.g.*, Luna/Solar [10] and Nitro SRD [28]) in the physical domain of CSPs. In particular, a Fisc agent extends Luna/Solar network stack and aggregates multiple network connections of Fisc clients on the same computation server. This substantially reduces the CPU and memory resources each client needs to reserve for network-related operations.

Offloading security-related functions to Fisc agent. We adopt an early-checking design to perform security checks (*e.g.*, authentication and authorization) in Fisc agent when it receives requests from Fisc clients. Different from the methods with network gateway, which deal with the malicious traffic at their proxies, this design prevents malicious traffic from consuming the resources of backend storage clusters.

Offloading storage-protocol functions to Fisc proxy. We choose to offload storage-protocol functions to Fisc proxies in the storage clusters, instead of Fisc clients, for three reasons. First, the DPU in the computation server has limited resources. After spending resources on network-related functions, security-related functions and bare-metal virtualization of virtio-Fisc device, the DPU does not have sufficient resources to implement complex storage protocols. Second, offloading these functions to the storage clusters helps move the storage traffic between the computation servers and the storage clusters in the backend network within storage clusters, saving the scarce network resources in the compute-storage disaggregated architecture. Third, it allows us to adopt storage-oriented optimization and hardware-assisted accelerations in the storage clusters to improve the overall system performance and reduce costs.

4.1.2 Simplification and Compatibility

Implementing an FS client and maintaining its compatibility across different versions of its software library is challenging because a typical FS client has a large number of APIs (*e.g.*, the HDFS client has more than 100 APIs [29]). We introduce a unified RPC-based method to simplify the implementation of Fisc clients and a mechanism similar to protocol buffers for compatibility maintenance.

Simplifying client implementation using RPC. We implement the APIs in the Fisc client using RPC stubs. When the application invokes an API, the Fisc client passes the parameters of the API to its corresponding RPC stub. The stub encodes these parameters, the file handle, and the tenant information into an RPC request. This request is sent to the Fisc agent in the DPU with a virtio-Fisc device (§4.3.1). The Fisc agent checks the authorization of tenants and looks up the file handle in its route table (§4.2.1) to forward the RPC request to a corresponding Fisc proxy. Upon receiving the request, the Fisc proxy resolves it and invokes the corresponding RPC service of the API, which completes the API and encodes its return value in an RPC response. The response is returned to the Fisc client along the opposite path of the RPC request and resolved by the client. This design makes it easier to implement and add APIs in Fisc clients.

Maintaining compatibility using a PB-based mechanism. Building on top of the RPC-based API implementation, we introduce a PB-based mechanism to maintain the compatibility of Fisc clients across different versions. Directly applying the PB protocol [30] would introduce extra data center tax of (de)serialization [31], wasting the limited resources in DPU. To this end, we categorize Fisc APIs into data-related ones (*e.g.*, read and append) and meta-related ones (*e.g.*, create, delete, open and close). Although the former has fewer APIs, it is more frequently used than the latter. Thus, for data-related APIs, we adopt several carefully designed, efficient data structures to maintain their compatibility. For the meta APIs, we use the PB protocol as it is. In this way, we can achieve a balance between performance and compatibility.

4.2 Storage-aware Distributed Gateway

SaDGW is a distributed gateway that sets up direct connections, referred to as "direct highways" in the paper, between the Fisc agents and the Fisc proxy. As such, Fisc can adopt high-performance network stacks on these direct highways, and further leverage storage semantics to build a file-granularity storage-aware routing. It improves the availability through storage-aware failure handling and improves the read throughput through locality optimizations.

4.2.1 Direct Highway Between Agents and Proxies

Direct highway. With the help of DPUs, Fisc builds direct highways between Fisc agents and Fisc proxies, where no network gateways are needed. Considering a storage cluster with thousands of nodes, this would be a significant cost saving. On the highways, we adopt high-performance network stacks of Luna/Solar [10], which is transparent to cloud-native applications, instead of the TCP/IP stack. Raw data structures [32] are adopted to eliminate the overhead of (de)serialization between Fisc agents and proxies.

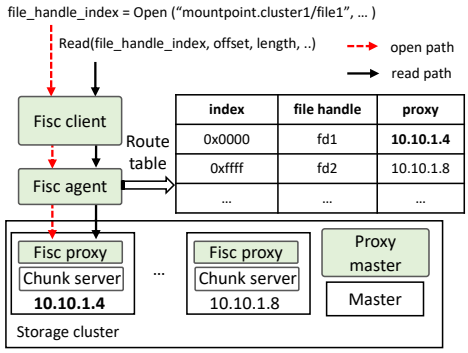


Figure 3: The routing process of Fisc.

File granularity route table. SaDGW manages highways through a centralized control mechanism (§3.3). As shown in Figure 3, Fisc agent adopts a file-granularity route table for routing file requests to Fisc proxies, which records the file handle information and the location of the Fisc proxy serving the file. For the route table, one entry is inserted once a file is opened for the first time. When Fisc agent receives a file open request, it randomly chooses a Fisc proxy from the proxy mapping. An entry is constructed when a response of successful file open is returned. The entry includes the returned file handle, the location of the chosen proxy, and the SLA-related attributes mentioned below. Afterwards, when an I/O request arrives at the Fisc agent, it looks up a proxy in the route table with the file handle of the request and then transmits the request to the proxy. Due to the scarce memory in DPU, Fisc uses an LRU policy to control the size of the route table.

4.2.2 Storage-aware Failure Handling

Enhanced route entry. Based on the file granularity route table, Fisc further leverages storage semantics to improve its availability. For failure handling of storage protocols, three main factors are considered: *retry timeout*, *retry destination*, and *highway quality*. 1) The *retry timeout* means the maximum number of times the Fisc agent retries the failed requests, which is related to the request timeout set by users and highway quality; 2) The *retry destination* denotes the proxy location in the entry, which will be replaced by a new proxy if retry timeout occurs; and 3) The *highway quality* is measured by the average latency to estimate the network quality to the proxy. Therefore, we enhance the route table to support failure handling. Besides the file handle and proxy location, each route entry is extended with three items: *retry times, retry timeout*, and *avg-latency*, which record when the agent reset the connection, the condition under which the agent gives up, and the average latency of requests, respectively. We make use of these items to implement storage-aware failure handling.

Failure handling. Fisc leverages several mechanisms in the Fisc agent to conduct failure handling.

- *Retry.* When detecting a failed request, the Fisc agent retries the request several times until it receives a successful response or it exceeds the timeout defined by users. Since users usually set a relatively large timeout for their requests, the Fisc agent initially sets a small empirical timeout (*i.e.*, ten times the average latency) to detect failed requests. When such a request is found, the agent doubles the timeout to execute the retry. This mechanism deals with temporary failures (*e.g.*, network jitters and burst proxy load).
- *Blacklist.* Upon detecting consecutive failures of requests or an abnormally large average latency to a Fisc proxy, the Fisc agent puts this Fisc proxy into the blacklist. A background thread periodically pings these proxies and will remove the successful pinged proxy from the blacklist. The metadata requests in the metadata path will exclude the proxies in the blacklist when choosing Fisc proxies. The data operations in the data path will involve the following reopen mechanism.
- *Reopen.* If the destination Fisc proxy of a request is in the blacklist, Fisc agent will select a new Fisc proxy to reopen the file and update the route entry. Otherwise, for a failed request, it adopts a threshold of retry times to make sure that there is still time left after the retry. In the remaining time, it reopens the file by retrying the request to a new Fisc proxy. This operation provides the opportunity to complete the request with the new proxy and avoid request failure.

These mechanisms are transparent to cloud-native applications. It provides flexibility for CSPs to upgrade the failure handling policy and helps keep the Fisc client lightweight.

4.2.3 Locality-aware Read

For a read operation, its request is first sent to a Fisc proxy and then sent to the Pangu chunkserver where the data to read is located by the proxy. The read response with the data to read is returned along the opposite path: from the chunkserver to the proxy and then to the client. It results in a two-time amplification of the read traffic, which consumes extra bandwidth and reduces the read throughput of the whole cluster by half. Considering that each storage node is deployed with a Fisc proxy process and chunkserver process, we design the locality-aware read by letting Fisc agent record the locations of file chunks and sending read requests to the proxy, where the concurrently deployed chunkserver holds their required chunks.

Predicted locations in a range table. When an open or read response returns to Fisc agents, the location information of the file chunks is piggybacked, as shown in Figure 4. The proxy returns the chunk information that would be read in the near future by the read prediction mechanism of Pangu. The number of the predicted chunks is empirically set to 16. Then the location information is encoded as range and location pairs and inserted into a range table. Each entry of the range table corresponds to a file, and the total number of range pairs in an entry is limited to 64 due to the scarce resource of DPU. For the 64 MB chunk size, its spanned range is 4 GB, which covers a large range space of files. The index of a file's corresponding range table entry is stored as a read hint attribute in a route table entry.

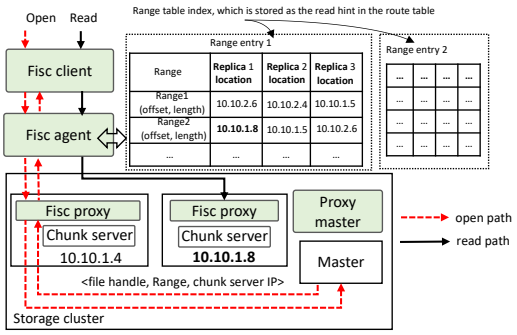


Figure 4: The design of locality-aware read.

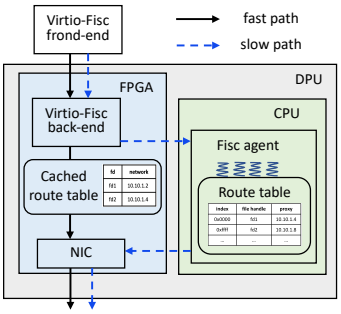


Figure 5: The design of fast path.

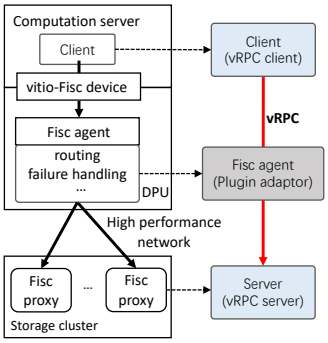


Figure 6: The abstracted vRPC.

Shared memory instead of cross-node communication. When a request arrives at a Fisc agent, the Fisc agent looks up its route entry and finds the read hint, which is an index of the range table. With the index, the Fisc agent then looks up the range table and finds the matching range and location pair. If a pair hits, the read request will be sent to the location in the pair. In some cases, the range for a read, with the offset adding the length to read, is larger than the range of a hit pair. In this case, due to the limitation of CPU resources, we do not divide a read request into multiple ones to avoid complex processing in DPU, such as segmentation, combination and failure handing. When the Fisc proxy of the location receives the read request, it calls the Pangu client to complete the request. As the Pangu client finds that the chunkserver and the proxy is located in the same physical node, it uses shared-memory communication instead of the network. As a result, the data to read is only transmitted once through the network, and increase the total read-throughput of a storage cluster.

4.3 SW/HW Co-design with DPU

Fisc adopts X-Dragon DPU [33] to build a novel virtio-Fisc device to accelerate its secure passthrough from the virtual domain of users to the physical domain of CSPs. To meet the requirements of cloud-native applications, a fast path is applied in DPU, and many optimizations are adopted to mitigate the impact of the scarce resources of DPU.

4.3.1 DPU-based Virtio-Fisc Device

The virtio-Fisc device is a PCIe device following the virtio standard. It consists of two parts, the frontend in VMs/containers and the backend in DPUs. Fisc client puts requests in the virtio hardware queues through the frontend, and Fisc agents running on the processor of DPU process the requests of the hardware queues. Agents send the requests to the Fisc proxies, and put the returned responses into the virtio hardware queues, which are consumed by the frontend. Two generations of virtio-Fisc devices are adopted in Fisc:

Virtio-Fisc devices based on virtio-block. We adopt virtio-block device for its compatibility with major operating systems and can be used by most VMs/containers without modification. With virtio-block interface, the front-end is the same as the standard virtio-block device, and a lightweight communication library is implemented with block read and write operations for Fisc client. However, the backend is quite different, and requests in hardware queues are processed with Fisc agents instead of the traditional virtio-block software, as mentioned in [20]. In fact, it only makes use of the virtio-block interface and works as a virtio-Fisc device.

Virtio-Fisc devices based on customized design. We design a novel vitio-Fisc device to eliminate the limitation of virtio-block. For example, the depth of a virtio-block queue is limited to 128 in most operating systems. Though it is enough for virtio-block but not for nonblocking requests of Fisc. The novel virtio device is more like a NIC device. We further leverage its interface to the RPC level, which can be suitable for cloud-native services like FaaS. It makes use of virtio queues to transmit commands for RPC requests and receive responses. We equip the device driver in our released OS in Alibaba.

4.3.2 Fast Path

We adopt a cache of route table in FPGA of DPU, which generates a fast path to speed up the processing of file requests. As shown in Figure 5, the mapping between the file handle and network connection is cached in the FPGA. With the cache, when a request with its file handle comes to the customized virtio-Fisc device in the FPGA, the FPGA resolves the file handle from the request and looks up the table. If it hits, the request will be directly packed as network packets and transmitted to the network connection. Otherwise, the request will be sent to Fisc agents via the slow path. Entries of the cache are controlled and updated by Fisc agent in software to relieve the complexity of the FPGA implementation of the cache. And to control network transmission bandwidth, the transmission window of each connection is also set and updated in cache entries by the Fisc agent.

4.3.3 Resource Optimizations

With regard to the scarce resources of DPU, optimizations for its CPU, memory and network are applied.

CPU optimization. We leverage two methods to optimize the CPU usage in DPU. 1) *Batch operation.* Fisc gathers multiple requests into one to share the processing of virtio protocols between Fisc clients and agents. 2) *Manual PB (De)Serialization*. Fisc adopts manual PB (De)Serialization methods. They are customized for particular data types of

Fisc and are more efficient than compiler-generated ones. According to our experiment with the manual methods, the IOPS can be improved by about 1.5% for 4 KB requests with one processor core of a DPU.

Memory optimization. The route table and range table occupy most memory in Fisc agent. To save memory, Fisc compresses the memory space of their entries. As the number of storage nodes is less than 1 million, we adopt 20 bits to represent the location IP instead of 32 bits, i.e. 4 bytes, in general. For locations of 3 replicas for a chunk, it consumes 8 bytes in total. A file with 64 predicted chunks for locality-aware read occupies 512 bytes. Taking file handle and tenant information into account, the total size of the memory space for a file in Fisc agent is no more than 1 KB. Thus, 1 GB of memory can hold up to 1 million files.

To further save the memory, we pass the range table to Fisc client as a hint. In this way, there is no need to store a large number of locations for locality-read in the range table in DPU. Instead, they can be stored in Fisc client. Fisc client is aware of the range of chunks and can find the location corresponding to its read request. When Fisc client sends a request, it is accompanied by the hint. Then Fisc agent first checks the file handle and tenant information with the route table in DPU. For a passed request, Fisc agent sends the request to the Fisc proxy according to the hint. For security, the location hint passed to Fisc client is encoded with an index and has no meaning to users. To avoid applications changing the hint maliciously, the index is checked in Fisc agents and Fisc proxies. If the check fails, the locality-read mechanism for the tenants will be forbidden for a period of time in Fisc agent. It then falls back to using the route table of fewer locations in DPU. Thus, with the hint, Fisc can save a lot of memory and support more range table entries in DPU.

Network optimization. SaDGW carefully deals with the number of connections for the direct highway. First, it adopts the shared-connection mechanism [13] to reduce the connections between Fisc agents and Fisc proxies. Second, it recycles the resources of network connections by periodically tearing down idle connections. Third, for the narrow inter-region Tbps-level network bandwidth compared to that of intra-region, Fisc agents only connect parts of proxies in different regions where there can be thousands of storage nodes. In this way, it is sufficient for inter-region network throughput and reduces the number of connections.

4.4 Large-scale Deployment

Fisc carefully deals with ease of use, load balance, and QoS to support applications running over 3 million CPU cores.

4.4.1 vRPC

As shown in Figure 6, we abstract a vRPC service from Fisc. It is similar to the traditional RPC mechanism of RPC client and RPC server. Clients placed in containers call an RPC request by vRPC stub in Fisc client, and the request is processed by the vRPC Server in Fisc proxy. For a cloud-native service, developers only need to concern the RPC stub for clients in containers and its RPC service for servers in the backend clusters. The implementation details of vRPC such as virtio device and SaDGW, are transparent to both clients and servers, which is different from the traditional RPC as follows. First, it provides a secure passthrough from the virtual domain to the physical domain with an efficient hardware-assisted virtio device. Second, its RPC request can be retried in Fisc agent, which is transparent to vRPC client, and high-performance network stacks can also be transparently adopted. Third, it gives an opportunity to adopt an adapter for a service, which can be integrated into Fisc agent to improve the availability of the service by its developers. vRPC can not only support Fisc FS but also other cloud-native services.

4.4.2 Load Balance

Fisc introduces two mechanisms for the load balance among thousands of Fisc proxies in storage clusters.

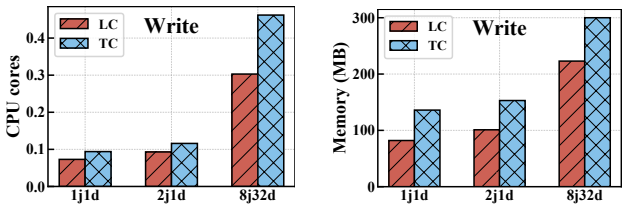
File granularity schedule. Traditional load balance relies on connection-based scheduling of network gateway, which focuses on balancing the number of network connections among proxies. However, a gap exists between the number of network connections and that of files. This means the number of connections may be balanced, but that of files in each proxy may be significantly different. To tackle this problem, Fisc eliminates the gap and presents a file-granularity schedule for load balance. Fisc agent forwards each file to a random Fisc proxy according to the hash value of its file name and other information such as access time. In this way, the files are evenly distributed among Fisc proxies. With locality-aware read optimization, as the chunks of files are evenly distributed to each data server by Pangu, it leads to an even balance of read requests to Fisc proxies, which are currently deployed with data servers in storage nodes.

The centralized re-scheduling. As Fisc proxy masters periodically collect the load of each proxy, they can schedule and migrate part of the files from a high-load Fisc proxy to a low-load one. The migration is transparent to the applications of tenants, and Fisc agent accordingly reopens these files after the migration. Meanwhile, Fisc proxy masters also push the load information to Fisc agents. After receiving the information, Fisc agents reduce the hash weight of high-load Fisc proxies and improve that of low-load ones. In this way, Fisc implements a centralized re-scheduling.

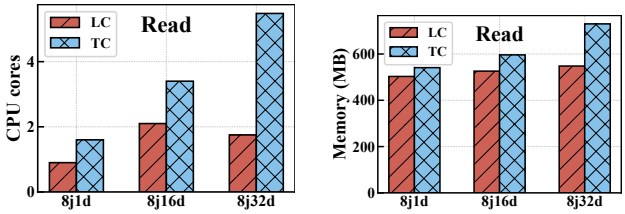
4.4.3 E2E QoS

Fisc supports hybrid file access for online real-time applications and offline batch-processing applications, the demands of which are represented by high priority and low priority.

Hardware-based QoS. Vitio-Fisc devices, NICs, and networks adopt hardware-based QoS mechanism. Virtio-Fisc devices and NICs make use of their hardware queues of high and low priorities. We set DSCP values in the IP packet header through the networking library to leverage the priority queues of network switches.



(a) Client's CPU utilization. (b) Client's memory consumption.



(c) Client's CPU utilization. (d) Client's memory consumption.

Figure 7: The resource consumption of LC and TC when writing/reading data to/from the storage cluster.

Software-based QoS. Fisc client, Fisc agent and Fisc proxy adopt a software-based QoS mechanism, utilizing a hybrid thread model of exclusive threads for high-priority and low-priority requests, respectively. The reason for the hybrid thread model is to avoid head-of-line (HOL) blocking problems. To conserve the scarce CPU resource of DPU, a large offline request is not divided into separate smaller ones to avoid the complex request combination and failure dealing. As a result, if high and low priority requests are in the same thread, there may be a HOL blocking problem between them. And the other reason is the lack of cache isolation capability of NIC like CAT for Intel CPU [34]. The buffer cache of NIC may be full-filled with low-priority packets if a DPDK-based polling thread [35] stops polling network packets. Therefore, if high and low priority network packets are processed in the same thread, the low-priority queue of NIC should keep polling. Otherwise, the buffer cache may be full-filled and eventually affect high priority traffic. However, the non-stop polling for low-priority requests in a thread makes it hard to guarantee the high-priority requests.

Besides Fisc modules, the backend Pangu also adopts software-based QoS for NVMe SSDs, as current SSDs lack of hardware-level QoS mechanism. Therefore, Fisc enables end-to-end priority classification.

5 Evaluations

We evaluate Fisc through extensive experiments in a testbed and demonstrate its performance in a production environment. We focus on the following measurements:

- The efficiency of Fisc lightweight client (§5.2).
- The performance of I/O requests in Fisc (§5.3).
- The availability of I/O requests in Fisc (§5.4).
- The impact of QoS scheme on multi-applications (§5.5).
- The effectiveness of load balancing in Fisc (§5.6).

5.1 Testbed Setup

Our testbed is a disaggregated cluster consisting of one computation server and a storage cluster of 43 commodity storage servers. The computation server is equipped with a DPU. The storage cluster is equipped with the Pangu storage system [6]. We use FIO [36] to generate different I/O workloads in the computation server and record the CPU and memory consumption of the client. The number of threads to issue I/O requests is denoted by $numjobs$. The number of inflight I/O is denoted by $iodepth$. For simplicity, we use $njmd$ in figures to represent the workload of $numjobs = n$, $iodepth = m$.

5.2 Lightweight Client

We first compare the resource consumption of Fisc client, denoted as LC, with that of a traditional FS client, which integrates the storage-related protocols (*e.g.*, three replicas) and network-related stacks (*e.g.*, RPC and TCP/IP) and is denoted as TC.

Microbenchmark. To test the resource utilization of clients with different data sizes, we write data to the file system with a granularity of 4 KB and read the file with a granularity of 128 KB. Figure 7 shows that LC has substantially lower CPU utilization and memory consumption than TC for both write and read operations. For example, when writing data with 8j32d (*i.e.*, $numjobs = 8$ and $iodepth = 32$), LC and TC each consumes 0.3 and 0.46 CPU cores, respectively. In another experiment where we let one FIO job write data to the storage cluster at a fixed rate of 1.75 GB/s, LC consumes less CPU and memory resources than TC by 69% and 20%, respectively.

Production environment. We also evaluate Fisc in a production system, which consists of thousands of servers and provides Swift service, a distributed streaming service similar to Kafka [37]. Figure 8 shows the bandwidth and CPU and memory consumption of Swift in one month when writing data to the remote storage cluster. Swift initially uses TC and switches to LC on day 18. After the switch, LC maintains the same high bandwidth performance as TC does, but consumes 16% and 57% less CPU and memory, respectively, than TC. Specifically, when we only offload erasure coding to Fisc proxies, the CPU and memory consumption of containers is reduced by 9% and 40%, respectively.

These results in the testbed and production environment demonstrate the efficiency and efficacy of the Fisc lightweight client in supporting cloud-native applications with high performance while consuming substantially fewer resources.

5.3 Latency

To evaluate the latency of I/O requests in Fisc, we first compare Fisc with a network-gateway-based load balancing solution [38], denoted as LB. We then validate the effectiveness of locality-aware read in the testbed.

Microbenchmark. We first start different FIO tasks on the computation server and measure the end-to-end latency of I/O

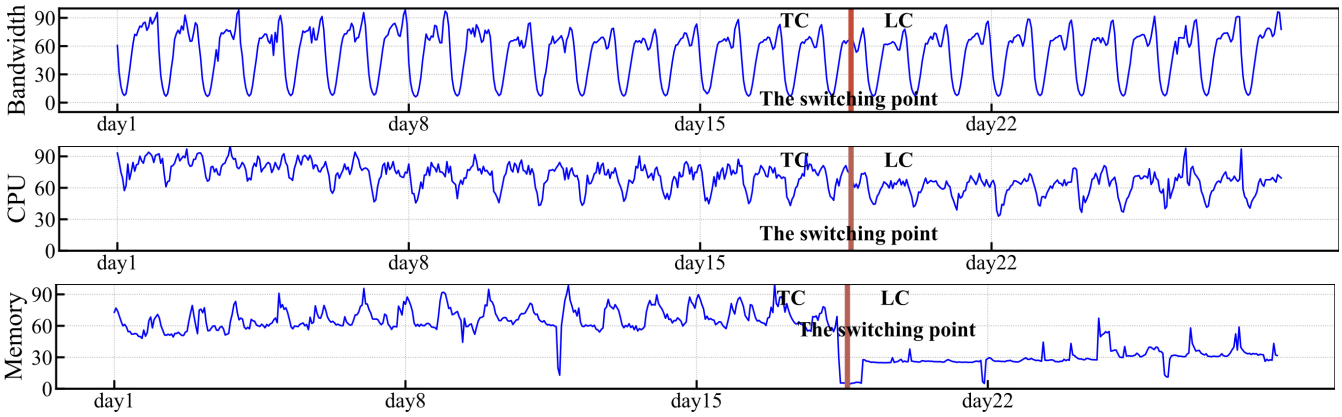
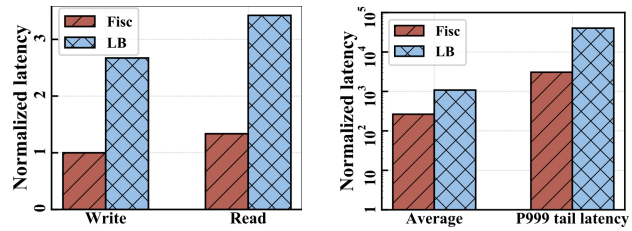


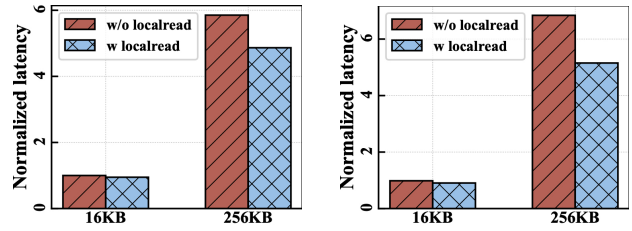
Figure 8: The bandwidth, CPU utilization, and memory consumption in a month. Results are given in the range of [0, 100].



(a) The write and read latency of a single job with a data size of 8 KB.

(b) The write latency with 64 jobs, a data size of 8 KB, and a fixed bandwidth of 100 MB/s.

Figure 9: The comparison of latency between Fisc and LB.



(a) The latency of sequential read.

(b) The latency of random read.

Figure 10: The effectivenss of locality-aware read.

requests. Figure 9(a) shows that the write and read latency of Fisc is 63% and 61% lower than those of LB when launching I/O requests with a data size of 8 KB. In the next experiment, we let Fisc and LB write files with 64 jobs and a data size of 8 KB at a fixed bandwidth of 100 MB/s. Figure 9(b) shows that Fisc reduces the average and P999 tail latency compared to LB by 76% and 92%, respectively. This latency improvement results from two optimizations: (1) the SaDGW provides one-hop communication instead of the two-hop communication in a centralized network gateway; (2) the SaDGW transparently adopts the high-performance networking stack to replace the inefficient TCP/IP stack.

To verify the benefits of locality-aware read, we further compare the latency of FIO tasks in sequential read and random read scenarios with a data size of 16 KB and 256 KB. As shown in Figure 10, the latency of read requests reduces in both scenarios (*e.g.*, by 25% when randomly reading files with a data size of 256 KB). It shows that the locality information in the range table is effective in helping route the read requests directly to the target storage server, reducing the end-to-end latency.

Production environment. We plot the average write latency of an online search workload over 30 days. As shown in Figure 11, the average latency is stable at ∼500 μs even when the workloads reach as high as millions-level IOPS. This result demonstrates that Fisc provides a low-latency file system service for cloud-native applications. In contrast, this latency becomes several milliseconds when the file system service is provided through LB.

5.4 Availability

We use the P999 tail latency as a key metric to measure the effectiveness of Fisc's storage-aware failure handling mechanisms in guaranteeing the availability of file system services.

Microbenchmark. To verify the impact of proxy failure on tail latency, we randomly kill some proxy processes in the storage cluster of 80 storage servers and record the tail latency for all I/O requests. As shown in Figure 13, we kill one proxy at t_1 and then kill five proxy processes at t_2. We observe that the tail latency increases to <40 ms for a short time and quickly returns to the previous level. This result shows that proxy failures in the storage cluster have a limited effect on the tail latency. It is because Fisc can retry the failed I/O requests with its storage-aware failure handling methods. As a result, such failures have a limited impact on applications.

Production environment. Figure 12 illustrates the P999 tail latency of online searching tasks over the same 30-day period. Most of the time, it stays under 30 ms. We analyze the spikes in the figure. The spikes t_1, t_2, t_3, and t_4 happen due to the upgrade of FS, at which we launch/stop some proxies in the storage cluster. Other spikes are caused by network jitters and storage node breakdowns. However, after each spike, the P999 tail latency quickly returns to a low level with the help of Fisc's storage-aware failure handling mechanisms.

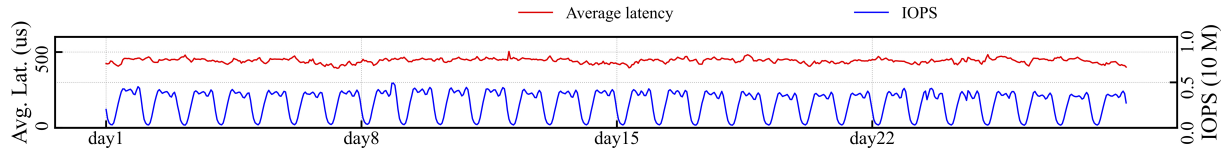


Figure 11: The average write latency and IOPS in one month.

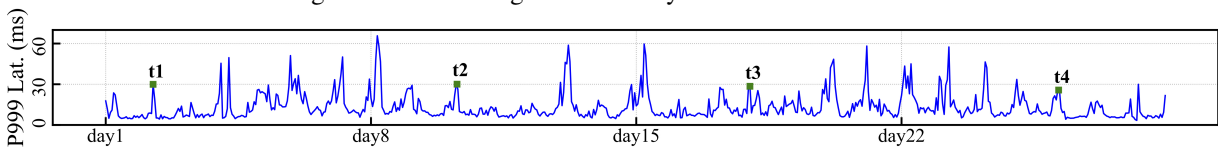


Figure 12: The P999 tail latency of write in one month in production environment.

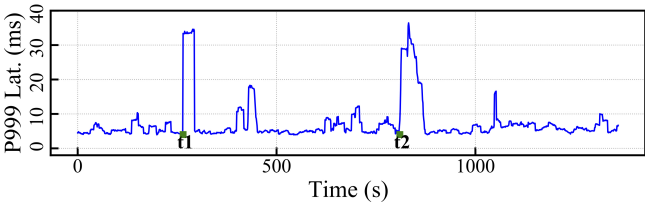


Figure 13: The P999 tail latency of write in micro-benchmark.

5.5 QoS

We demonstrate Fisc's ability to guarantee the QoS across different applications by measuring the latency of online searching tasks and the throughput of offline AI training tasks in a production environment. Both tasks are deployed in the same computation cluster and share the same storage cluster. Figure 14 shows that the latency of online search tasks stays stable and is barely affected by the fluctuated offline AI training tasks. It is because Fisc assigns a high priority to latency-critical tasks like online search and guarantees the corresponding QoS with an E2E QoS mechanism.

5.6 Load Balancing

To study the load-balancing capability of Fisc, We randomly choose six storage servers from our storage cluster and measure their normalized read IOPS over seven days. We compute the coefficient of variation of these nodes as a measurement of Fisc's load-balancing capability [39]. As shown in Figure 15, the coefficient of variation of read IOPS is < 5%. This result indicates that the read requests are evenly distributed among Fisc proxies and proves that Fisc achieves a similar quality of load balancing as Maglev, Google's in-house load balancer [39], whose coefficient of variation is 6-7%. This efficacy is due to Fisc's fine-grained storage-aware load-balancing strategy. Specifically, Fisc agents forward I/O requests with a granularity of files. In contrast, network-based load balancing methods forward I/O requests with a granularity of network connections, causing unbalanced numbers of files forwarded to different storage nodes.

6 Discussion

Not just migration. The two-layer aggregation of Fisc offloads network-related functions and storage protocols to Fisc agent in DPU and Fisc proxy in the storage node. The question is whether Fisc merely transfers the resource consumption from containers of users to DPUs and back-end storage clusters of CSPs but does not reduce the total amount of consumed resources. The answer is that Fisc not just migrates resources spatially but can significantly reduce resource consumption, because it *"aggregates"* the resource for storage protocols and network stacks processing in terms of tenants, applications and workloads. For example, one application in containers usually pre-allocate I/O threads and reserve memory and network connections, which cannot be shared with the applications in other containers. However, in Fisc, these resources are *"migrated"* and *"aggregated"* in Fisc agents and proxies, and they are efficiently shared by multiple applications to achieve high resource utilization. Furthermore, with function offloading, Fisc can leverage modern hardware-assisted acceleration for these storage protocols and advanced network-related stacks to improve their efficiency. For example, the Erasure-coding and CRC operation can be accelerated by hardware in the storage cluster.

With the development of cloud-native applications, more cloud-native services should aggregate their service-related resources among containers. Based on the traditional aggregation of VM resources, it will further improve the resource efficiency of CSPs.

Ecosystem service. The ecosystem is vital for cloud-native applications. Fisc extends its ecosystem in two aspects: compatibility with HDFS ecosystem and virtio-Fisc devices for different operating systems. For the former one, Fisc Client adopts a Java Native Interface (JNI) method to use its lightweight client of C language, and many optimizations have been introduced for the semantics compatibility between HDFS and Pangu. For the latter issue, we abstract virtio-Fisc devices to more general virtio-RPC devices, which are suitable for more cloud-native services. And we develop the virtio-Fisc driver in our released OS and will submit it to the open source community.

Resource in DPU. The resources in DPU are scarce, and Fisc also has to share these resources with other virtualization services, such as virtual networking and block services. Therefore, Fisc adopts a variety of optimization technologies to economize resource utilization, as mentioned in §4.3.3. With the development of DPUs such as Intel IPU [40] and

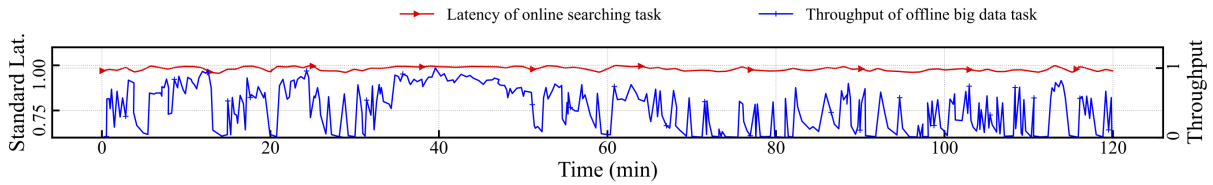


Figure 14: The latency of online tasks with background offline tasks in one month.

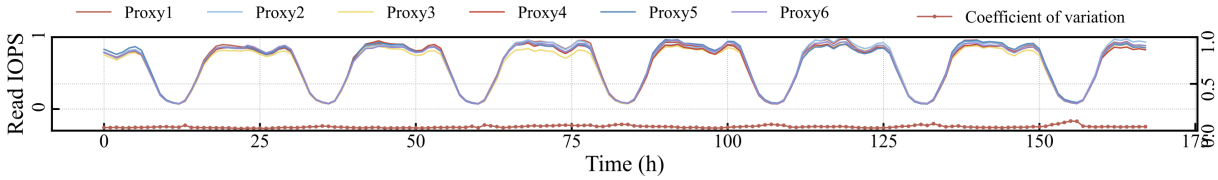


Figure 15: The load distribution of read IOPS of six storage nodes in one week.

Nvidia DPU [41], the processing capability of embedded processors of DPU has been greatly improved. Meanwhile, more hardware acceleration functions like compression have been integrated. These new features help Fisc agents adopt more complex policies to deal with failure handling and locality-aware read mechanisms. It is noteworthy that careful resource optimizations are still needed with the increase of throughput from 25 Gbps to 100 Gbps or 200 Gbps.

7 Related Work

Infrastructure support for cloud-native applications. Many studies [4, 42–51] have investigated how to provide efficient infrastructure support for emerging cloud-native applications (*e.g.*, microservice, container and serverless computing), including state management [43–45], runtime [46], data storage [47], fault tolerance [4] and performance optimization [48–51]. Some work [52,53] also looked into designing efficient service interfaces for cloud-native applications. For example, LogBook [52] provides logging interfaces for stateful serverless applications and uses a metalog to address log ordering, read consistency, and fault tolerance. Fluid [53] provides a unified data abstraction for cloud-native deep learning applications. In this paper, we design Fisc, a large-scale file system that provides high-performance file system services for cloud-native applications.

High-performance distributed file systems. Many distributed file systems have been designed and deployed (*e.g.*, pNFS [54], NAS [15], Facebook Tectonic [7], Google Colossos [5], and Alibaba Pangu [6]) to provide high-performance storage services for applications. However, they are ill-suited for cloud-native applications because they use heavyweight clients and a centralized network gateway. To this end, some studies (*e.g.*, OFC [55], FaaSCache [56], FLASHCUBE [57], and Pocket [58]) proposed adding cache to the persistence layer to improve the performance. However, they still suffer from a low level of resource multiplexing. In contrast, Fisc proposes the design of a lightweight client and storage-aware gateway, and resorts to a software-hardware co-design to provide high-performance file system services for cloud-native applications.

Bare-metal DPUs in clouds. The cloud computing community is increasingly developing and deploying bare-metal DPUs in clouds (*e.g.*, Nitro [19], BM-Hive [20], ELI [59], Splinter [60], and Bluebird [61]). Some studies also use DPUs to accelerate file system services (*e.g.*, LineFS [62], Gimbal [63], and Leapio [64]). However, they are not designed to provide cross-domain file system services between tenants and CSPs. In contrast, Fisc leverages the X-Dragon DPU in the computation server and introduces a new virtio device to provide secure, high-performance cross-domain file system services.

8 Conclusion

The trend of cloud-native brings new challenges and opportunities for CSPs to revisit their file system services. In this paper, we present Fisc, a large-scale cloud-native-oriented file system, which adopts a two-layer aggregation mechanism to multiplex resources of file clients among containers and a distributed storage-aware gateway to improve performance, availability and load balance of I/O requests. Fisc also adopts virtio-Fisc device with DPU for high performance and secure passthrough from users' virtual domain to CSPs' physical domain. Fisc has been deployed in a production DCN for over three years and provides large-scale file system service for cloud-native applications.

Acknowledgements

We are extremely grateful to our shepherd, Liuba Shrira, and the anonymous FAST'23 reviewers for their invaluable feedback. We also thank Yuxin Wang, Ridi Wen, and Haohao Song for their help during the preparation of the paper. Lulu Chen and Jie Wu are supported in part by the National Key R&D Program of China 2021YFC3300600 and an Alibaba Innovative Research Award. Qiao Xiang is supported in part by an Alibaba Innovative Research Award, NSFC Award 62172345, Open Research Projects of Zhejiang Lab 2022QA0AB05, and NSF-Fujian-China 2022J01004. Xiaoliang Wang is supported by NSFC Award 62172204.

References

[1] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *NSDI'19*, pages 193–206. USENIX Association, 2019.

[2] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards Demystifying Serverless Machine Learning Training. In *SIGMOD'21*, pages 857–871. ACM, 2021.

[3] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service Workflows. In *ATC'21*, pages 957–971. USENIX Association, 2021.

[4] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-Tolerant and Transactional Stateful Serverless Workflows. In *OSDI'20*, pages 1187–1204. USENIX Association, 2020.

[5] Google. A peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, 2022.

[6] Pangu. The High Performance Distributed File System by Alibaba Cloud. https://www.alibabacloud.com/blog/, 2022.

[7] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *FAST'21*, pages 217–231. USENIX Association, 2021.

[8] Amazon. AWS Elastic File System. https://docs.aws.amazon.com/efs/latest/ug/whatisefs.html, 2022.

[9] Microsoft. Azure Data Lake Storage Gen2. https://learn.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-introduction, 2022.

[10] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *SIGCOMM'22*, pages 753–766. ACM, 2022.

[11] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Wang, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. Deployed System: More Than Capacity, Performance-oriented Evolution of Pangu in Alibaba. In *FAST'23*. USENIX Association, 2023.

[12] Hdfs. Hadoop HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2022.

[13] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When Cloud Storage Meets RDMA. In *NSDI'21*, pages 519–533. USENIX Association, 2021.

[14] Alibaba cloud. Object Storage Service. https://www.alibabacloud.com/help/en/object-storage-service, 2022.

[15] Alibaba cloud. Apsara File Storage NAS. https://www.aliyun.com/product/nas, 2022.

[16] Alibaba. Maxcompute. https://www.alibabacloud.com/product/maxcompute, 2022.

[17] Alibaba cloud. The exploration of cloud-native. https://developer.aliyun.com/article/721889, 2021.

[18] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *ATC'22*, pages 53–68. USENIX Association, 2022.

[19] Amazon. AWS nitro system. https://aws.amazon.com/cn/ec2/nitro/, 2022.

[20] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density Multi-tenant Bare-metal Cloud. In *ASPLOS'20*, pages 483–495. ACM, 2020.

[21] Xiantao Zhang, Xiao Zheng, and Justin Song. High-density Multi-tenant Bare-metal Cloud with Memory Expansion SoC and Power Management. In *HotChips'20*, pages 1–18. IEEE, 2020.

[22] Yuyu Luo, Chengliang Chai, Xuedi Qin, Nan Tang, and Guoliang Li. Visclean: Interactive cleaning for progressive visualization. *Proceedings of the VLDB Endowment*, 13(12):2821–2824, 2020.

[23] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Bluesky: A Cloud-backed File System for the Enterprise. In *FAST'12*, pages 1–14. USENIX Association, 2012.

[24] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage*, 9(4):1–33, 2013.

[25] Yilong Li, Seo Jin Park, and John Ousterhout. MilliSort and MilliQuery:Large-Scale Data-Intensive Computing in Milliseconds. In *NSDI'21*, pages 593–611. USENIX Association, 2021.

[26] Amazon. AWS storage gateway. https://aws.amazon.com/cn/blogs/storage/deploy-a-highly-available-aws-storage-gateway-on-a-vmware-vsphere-cluster/, 2022.

[27] Digitalocean. DigitalOcean. https://www.digitalocean.com/community/tutorials/how-to-create-a-high-availability-setup-with-heartbeat-and-reserved-ips-on-ubuntu-14-04, 2022.

[28] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A Cloud-optimized Transport Protocol for Elastic and Scalable HPC. *Micro*, 40(6):67–73, 2020.

[29] Apache. HDFS APIs. https://github.com/apache/hadoop/blob/trunk/hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/fs/FileSystem.java, 2022.

[30] Google. Protocol Buffers. https://developers.google.com/protocol-buffers, 2022.

[31] Svilen Nikolaev Kanev. *Efficiency in Warehouse-scale Computers: A Datacenter Tax Study*. PhD thesis, Harvard University, pages 1–24, 2017.

[32] John Biddiscombe, Anton Bikineev, Thomas Heller, and Hartmut Kaiser. Zero Copy Serialization Using RMA in the HPX Distributed Task-based Runtime. In *Proceedings of the International Conference on WWW/Internet 2017 and Applied Computing*, pages 1–8. IADIS, 2017.

[33] Shuangchen Li, Dimin Niu, Yuhao Wang, Wei Han, Zhe Zhang, Tianchan Guan, Yijin Guan, Heng Liu, Linyong Huang, Zhaoyang Du, et al. Hyperscale FPGA-as-a-Service Architecture for Large-scale Distributed Graph Neural Network. In *ISCA'22*, pages 946–961. IEEE, 2022.

[34] Intel. Introduction to Cache Allocation Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, 2022.

[35] DPDK. DPDK Poll Mode Driver. https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html, 2022.

[36] Flexible i/o tester. Flexible I/O tester. https://fio.readthedocs.io/en/latest/, 2022.

[37] Apache. Kafka. https://kafka.apache.org/intro, 2022.

[38] What is ALB. Server Load Balancer. https://www.alibabacloud.com/help/en/server-load-balancer/latest/what-is-application-load-balancer, 2022.

[39] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI'16*, pages 523–535. USENIX Association, 2016.

[40] Intel. Intel® Infrastructure Processing Unit (Intel® IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, 2022.

[41] Nvidia. NVIDIA BlueField Data Processing Units). https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2022.

[42] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *ATC'19*, pages 475–488. USENIX Association, 2019.

[43] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help Rather than Recycle: Alleviating Cold Startup in Serverless Computing through Inter-Function Container Sharing. In *ATC'22*, pages 69–84. USENIX Association, 2022.

[44] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-As-A-Service. *arXiv preprint arXiv:2001.04592*, pages 1–15, 2020.

[45] Zhe Wang, Teng Ma, Linghe Kong, Zhenzao Wen, Jingxuan Li, Zhuo Song, Yang Lu, Guihai Chen, and Wei Cao. Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA. In *ATC'22*, pages 639–654. USENIX Association, 2022.

[46] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless Computing on Heterogeneous Computers. In *ASPLOS'22*, pages 797–813. ACM, 2022.

[47] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service. In *ASPLOS'22*, page 782–796. ACM, 2022.

[48] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *OSDI'22*, pages 303–320. USENIX Association, 2022.

[49] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: A Native Serverless System for Low-latency, High-Throughput Inference. In *ASPLOS'22*, pages 768–781. ACM, 2022.

[50] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *ASPLOS'22*, page 753–767. ACM, 2022.

[51] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *NSDI'21*, pages 653–669. USENIX Association, 2021.

[52] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *SOSP'21*, pages 691–707. ACM, 2021.

[53] Rong Gu, Kai Zhang, Zhihao Xu, Yang Che, Bin Fan, Haojun Hou, Haipeng Dai, Li Yi, Yu Ding, Guihai Chen, et al. Fluid: Dataset Abstraction and Elastic Acceleration for Cloud-native Deep Learning Training Jobs. In *ICDE'22*, pages 2182–2195. IEEE, 2022.

[54] Dave Hitz, James Lau, and Michael A Malcolm. File System Design for an NFS File Server Appliance. In *WTEC'94*, pages 1–23. USENIX Association, 1994.

[55] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. OFC: An Opportunistic Caching System for FaaS Platforms. In *EuroSys'21*, pages 228–244. ACM, 2021.

[56] Alexander Fuerst and Prateek Sharma. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *ASPLOS'21*, pages 386–400. ACM, 2021.

[57] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *PLOS'21*, pages 38–45. ACM, 2021.

[58] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI'18*, pages 427–444. USENIX Association, 2018.

[59] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. *SIGPLAN*, 47(4):411–422, 2012.

[60] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal Extensions for Multi-tenant Low-latency Storage. In *OSDI'18*, pages 627–643. USENIX Association, 2018.

[61] Manikandan Arumugam, Deepak Bansal, Navdeep Bhatia, James Boerner, Simon Capper, Changhoon Kim, Sarah McClure, Neeraj Motwani, Ranga Narasimhan, Urvish Panchal, Tommaso Pimpo, Ariff Premji, Pranjal Shrivastava, and Rishabh Tewari. Bluebird: High-performance SDN for Bare-metal Cloud Services. In *NSDI'22*, pages 355–370. USENIX Association, 2022.

[62] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *SOSP'21*, pages 756–771. ACM, 2021.

[63] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling Multi-tenant Storage Disaggregation on SmartNIC JBOFs. In *SIGCOMM'21*, pages 106–122. ACM, 2021.

[64] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. Leapio: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *ASPLOS'20*, pages 591–605. ACM, 2020.