PoV: Storage Disaggregated Service Provision for Large Scale Cloud-native Applications

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

Your abstract text goes here. Just a few facts. Whet our appetites. Not more than 200 words, if possible, and preferably closer to 150.

1 Introduction

Alibaba Cloud is one of the top cloud providers in the world. Since 2009, the year Alibaba Cloud was established, we have been building up the unified storage platform, Pangu, which provides scalable, high-performance, and reliable storage service for different kinds of applications, such as Alibaba Cloud, Taobao, Tmall, AtnFin, Alimama, and so on [1]. At the same time, driven by the advantages of cloud-native ecosystem, represented by container and microservice technology, cloud applications are increasingly shifting from monolithic services, to large numbers of loosely-coupled microservices. The cloud-native applications can launch thousands of light weighted tasks with high elascity and dynamicity, and these tasks can be scaled automatically based on the users' demands. It becomes evident that large enterprises will continue to deploy their workloads with microservices to improve their flexibility, elasticity, and modularity.

As cloud-native applications penetrate into various business scenarios. Cloud providers are facing new challenges of managing the infrastructure, running environment, status, data storage for applications. As a storage platform, Pangu also encounters several tricky problems when providing ubiquitous and consistent IO service. It is highly desirable for Pangu to adequately support and cloud native applications in an imperceptible way. Cloud native applications have several distinct characteristics compared with traditional applications. Such as fast startup, short lifespan, hybrid deployment and so on. However, existing resolutions are proposed for traditional monolithic applications, failing to provide fast and short-lived storage service with the lifespan of cloud native applications. In addition, the data access process from compute node to storage node demands high performance due to the performance constrains, such as the search business which is highly latency sensitive. Therefore, several challenges have to be cleared before providing ubiquitous cloud-native oriented storage service.

Light weighted and service-oriented. One key characteristic of cloud-native applications is that they are composed of many light weighted services, this characteristic guarantee developers can launch thousands of short-lived tasks instantaneously, and these tasks can frequently start, stop, scale, rollback according to developers' demands. For example, the startup time for the container in alibaba have been reduced to 110ms, and 2000 instances can be launched in one physical server [6]. The storage service must be light weighted as many cloud native applications are invoked rapidly to serve a user's query. The lifespan of each components ranges from tens of milliseconds to several minutes, which also requires a high invocation rate for storage service. Otherwise, the latency of tasks will be violated and the resources of the compute nodes will be wasted due to the stagnation storage accessing process. At the same time, motivated by the decoupled architecture, software engineers can update and implement their code independently, frequently and flexibily. Companies deploy their services 100-1000+ updates per day [4]. Thus, service-oriented storage resolution is expected to provide consistent storage service.

Performance. High performance must be guaranteed by our storage service, such as low IO latency, high throughput. Many companies have implemented cloud-native techniques, they can respond to market conditions rapidly. Among these services, many are latency critical, such as the instance messaging [30], interactive online services [29], and online data intensive applications [25]. At the same time, the execution time of functions in cloud-native environment has unique feathers compared with traditional applications. Take Microsoft Azure's functions traces as an example [8], 50% of the functions execute for less than 1s on average, 50% of functions have maximum execution time shorter than 3s [24]. Storage service must guarantee the demands of these short lived and high performance microservices, otherwise the performance of applications well be deteriorated. A wealth of research have

attempted to provide performance aware storage performance storage for cloud-native applications [23] [20] [18] [17].

QoS. To make cloud native computing ubiquitous is a mission for many researchers and organizations [5]. On the computing side, organizations are developers can run their applications in dynamic environments such as public, private, and hybrid cloud. To provide ubiquitous storage service for these applications, as a storage pool, Pangu bares great pressure in guaranteeing performance isolation and QoS of different cloud native applications. Pangu not only supports the data access of alibaba's core business, but also satisfies the requests from the public tenants. Multiple business built on Pangu inevitably results to the resource competition, such as online tasks and offline tasks will seize the storage resources. What's more, diverse workloads focus on different performance metric during their lifetime, for example, online search workloads are latency critical, offline recommendation workloads are throughput critical. These workloads are running in the compute cluster and their storage service are offered by Pangu. At the same time, as the scale of applications becomes larger and larger, the availability and stability of storage service system are facing new challenge. Any disruption in the data access process ends up with applications' downtime and users' dissatisfaction. A sophisticated storage service system must guarantee the performance isolation of diverse applications and satisfies their diverse QoS requirements.

Solving the aforementioned challenges is nontrivial for storage service system. The cloud native applications are dynamic, flexible, and diversify, requiring corresponding storage service. The storage access process is complex, handling highly structured data with sophisticated protocol stack, including network protocol and storage protocol. Thus, we are motivated to propose a innovative cloud native-oriented storage service system based on Pangu.

In this paper, we present PoV, a storage service system for large scale cloud native applications in disaggregated cloud storage system. We describe the challenges in our design. PoV achieves the goal of providing lighted weighted storage services by imposing a light weighted API that hide the more complex storage protocol and network protocol. What's more, PoV provides high performance storage service by the design of offloading heavy protocols on a self-depolyed and dedicated hardware MoC (Microserver On Card) located at the compute node, which not only accerates the data processing progress but also promotes the security of storage provision system. In addition, a full path QoS mechanism is leveraged in PoV to ensure ubiquitous storage service provision. Finally, We show that PoV can solve the aforementioned challenges by leveraging it in real production system and evaluated its effectiveness by running large scale cloud native applications. We shared our experience in implementing PoV in large scaled practice.

2 Background

The development of cloud native technologies empowers more and more organizations to build and run their applications in dynamic environments such as public, private, and hybrid clouds flexibly [5], including video processing [21] [27], data analytics [22], machine learning [16], database services [11], and transactional workflows [19] [28]. The increasingly deployed cloud native technologies free developers from the burden of managing their infrastructures, enabling developers to concentrate on building the applications. The cloud service provider are responsible for providing infrastructure and managing the running environment for these applications. Currently, these applications are mainly developed in containers or serverless. Due to the stateless nature of these cloud native applications, their requested data, intermediate data, and output data is saved through remote storage, such as Pangu. Unfortunately, the storage access service are lagging behind and fails to satisfy the cloud native applications' requirements in terms of performance, security, and QoS.

We provide an overview of the architecture of Panguand give a brief introduction of server in alibaba's cloud native platform(2.1). We then present the unique characteristics of cloud native applications and their requirements (2.2). Finally we summarize and analyze the limitations of existing data path in disaggregated storage (2.3).

2.1 Pangu and Cloud Native Platform

Pangu is a distributed storage system which provides stable, scalable, and high-performance storage services. Diverse cloud storage services such as EBS, OSS, NAS, etc. are built upon Pangu and have served enterprises and developers all over the world. Now, Pangu has been the storage middle platform of Alibaba supporting all core businesses such as Taobao, Tmall, AntFin, DingDing, and Alimama, etc. At the same time, Driven by the great advantages of cloud native technologies, Alibaba also moved its core business to cloud native platform, which was built on X-Dragon bare metal servers. X-Dragon servers offload the storage and network virtualization overhead to a hardware accelerator card called MoC, thus reducing the computing virtualization overhead in the server [9]. MoCs have their own compute, memory, operating system, and kernel, allowing them to allocate resources to network and storage services.

2.2 Cloud Native Applications

Cloud-native applications are born in cloud, and they are composed of many light weighted tasks, always running with function as a service (Faas). This characteristic guarantee developers can launch thousands of short-lived tasks instantaneously, and these tasks can frequently start, stop, scale, rollback according to developers' demands. In particular, the

cloud native applications have some unique features compared with traditional applications.

Fast startup. The light weight feature enables the services in cloud native applications to invoked and startup rapidly to user's demand. Unlike monolithic applications, the startup time can even achieve millisecond level for some functions [12], and cloud providers are still devote to reduce the startup time to provide extremely low latency service for some applications, such as online searching.

Short lifespan. The lifespan of microservices or functions in a cloud native application ranges from tens of milliseconds to several minutes. Due to the light weight feature, most of them are short-lived, especially latency critical applications, such as interactive online services [29]. For example, on microsoft's cloud native platform, 54% of containers' lifespan do not exceed 5 minutes [8]. The lifespan of search service in Taobao is also minute level.

Update frequently. Cloud native applications are designed to be elastic. They can handle failed components or update their components gracefully and independently due to their design of decoupling. This enables developers update their applications frequently, for example, Uber's applications are deployed several thousand times each week [4].

Diversity workloads. The light weight nature of cloud native applications allow many workloads to share a compute and storage cluster. Diverse workloads are glad to embrace cloud native computing. In addition to traditional applications such as web servers and databases, many new applications in other fields like bioinformatics, data science, and high performance computing are also split into many components.

Flexible and salable. The decouple designing of cloud native applications make each components scale up and shrink flexibly. Besides, cloud providers can adjust its resource allocation strategy dynamically to copy with the cloud native applications' demand, which highly improve the efficiency of resource utilization.

2.3 Data Path

In the era of cloud computing, we have observed that enterprises are reshaping their architectures to embrace the benefits of cloud native applications, including BigData, Middleware, Database, Security, and Bare-metal Servers. This new paradigm free tenants from the burden of managing the servers. The cloud native applications is stateless, which means their state, their access to data have to be managed by others, such as cloud providers. Thus, the state of cloud native applications and their IO data are persisted in remote storage, such as Amazon S3 [2], Microsoft Azure Blob Storage [7], and queues [3]. It has the benefits of seperating compute and storage resources when load data in remote storage, but significant performance overhead occurs compared with local memory storage [18] [15]. However, it inevitably causes high cost when loading cloud native applications' data in memory. Thus, the data path from compute node to storage cluster is urgently needed. Existing data path in Pangu have limitations. Directly data access through the network between compute and storage cluster inevitably causes security risks to the storage cluster, i.g. expose the IP address of storage server. And existing resolutions implemented as network middle layer to respond storage IO requests and forward these requests to certain storage server, such as Load balancers. However, the obvious drawback of the network solution is the performance degradation for data access. The trade-off between security and performance poses new challenge in designing storage provision system.

All in all, current cloud providers and their data path fail to simultaneously provide low latency, high throughput, and high scalability storage services [18] [22].

3 Motivation

3.1 Challenges

With the gradual popularization of cloud native scenarios, containers are favored by more and more applications because of their lightweight, high isolation, and easy management. For example, the computing end of alibaba's search, advertising and other businesses is mainly operated in the form of containers and bare metals. However, in the context of computing storage separation, how to access storage clusters in containers with performance requirements similar to or even better than accessing local storage is an important guarantee for cloud native to be accepted by consumers. In the process of realizing this goal, we are faced with some problems that cannot be ignored.

Cloud native lightweight requirements.Traditional distributed storage systems, such as ceph and Pangu, require more functions on the client side in order to serve storage access. In addition to the regular data access functions, we also need to deal with the implementation of EC, backup, fault node avoidance and other logical functions. These functions consume resources and memory on the client side. In cloud native environment, in order to adapt to rapid deployment, easy to upgrade the characteristics of the iteration, the original large-scale programs will be split into different parts according to the logical function, and are running in different containers. If accessing to the storage system logical function of each container are very heavy and complicated resource-intensive, container deployment speed will be slow. What's more, there is also no way to run multiple containers on a single host. On the other hand, upgrading some of the functionality on the client side would be an enormous task, as we would need to modify the contents of thousands of containers. Traditional distributed file systems cannot meet the requirements of lightweight and low resource consumption in the cloud native environment, nor can they be upgraded quickly.

Performance. In the cloud native environment, there are a

variety of mixed systems. For example, Alibaba's ODPS is a bandwidth sensitive application and search is a latency sensitive application.We need to design a storage system that can provide high bandwidth for some applications and low latency for others in the same cluster, rather than deploying different storage systems with different features and configurations to meet different applications' requirements. On the other hand, applications deployed using containers have difficulty taking advantage of hardware acceleration due to the virtualization and isolation nature of containers. Although the virtualization technology of RDMA has matured in recent years to enable applications in virtualized containers to use high-performance networks, it is still a huge engineering effort to modify current applications to fit the RDMA programming interface. Therefore, how to ensure that the storage system can meet the performance requirements of different applications without changing the applications is a problem that needs to be solved.

Service diversity and storage pooling.In cloud native scenarios, different types of services run by different tenants may run on the same host at the same time. These services may be latency-sensitive or bandwidth-sensitive. When these applications need to access the back-end storage at the same time, how to determine the priority of the different service to ensure the QoS of the application is also a problem to be solved for the back-end storage service. In addition, multiple storage cluster may be in the different physical locations, when all cluster resources are pooled into a resource pool, how to make the front end application in accessing the back-end storage cluster of any resources, is not affected by geographical location, or other factors, and can achieve similar performance, is also a problem worthy of studying.

3.2 Choice

The way to access resources in back-end distributed storage (storage cluster) for applications running in virtual machines or containers is a problem worth discussing. There are two main solutions in the industry.

Usually, a computing cluster and a storage cluster are not in the same VPC(virtual private cloud). As shown in figure 1, to access data in another VPC, all traffic from the compute nodes are forwarded to the gateway and then through the load balancer to the back-end storage cluster. After the data reaches the back-end storage node, unfortunately, the load balancer does not know the actual storage location of the data. Therefore, the data need to be forwarded again before it reaches the real data storage node for read/write operations.

Although this is a common solution in the current cloud storage environment, the solution of connecting computing cluster and storage cluster through gateway has the following three problems that cannot be ignored:

- Complex clients slow deployment, occupy resources, and cause difficult upgrade. The solution based on the network layer has complex processing logic on the client side. It not only deals with basic logic such as connecting and transferring data, but also deals with storage protocol-related processing, such as EC, backup logic and the error processing logic when the front-end storage performance decreases due to the failure of the back-end storage node during the access to the storage cluster. What's more, It also suffers from the complexity of the upgrade described in section 3.1.

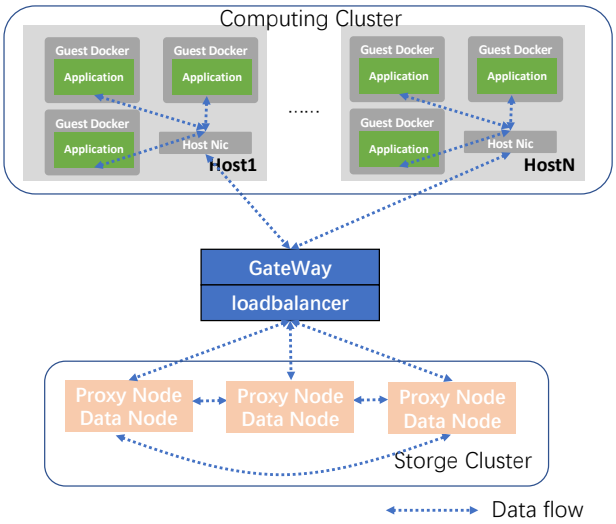

Figure 1: gateway

- The centralized gateway becomes the bottleneck of performance improvement. As shown in figure 1, all traffic of front-end applications passes through the gateway and is forwarded to the back-end storage cluster after load balancing. Therefore, it is required that the gateway can handle and forward massive traffic, and at the same time can guarantee the demand of high bandwidth. For example, in a computing cluster with 1000 nodes, the bandwidth of each node is 50 GB. To prevent the bandwidth loss of each node, the bandwidth of the gateway node is not less than 50 TB. This kind of large-scale gateway cost too much and is not economically friendly. Especially in the case of rapid service growth, to ensure high bandwidth and low latency for front-end applications to access storage, centralized gateways may become the bottleneck of storage services. As the number of compute nodes and the bandwidth of each node increases, such high-performance gateways are not even possible. Therefore, the scale and bandwidth of gateway machines need to be expanded in time, which is a huge cost in enterprise-level applications.

- Increased the traffic in storage cluster. All traffic in the computing cluster must be forwarded by the gateway before reaching the back-end storage cluster. The gateway

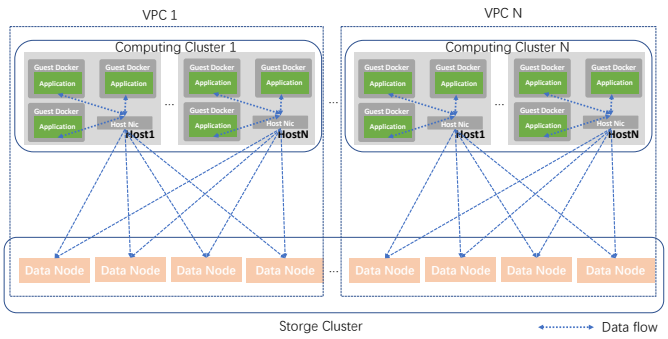

Figure 2: VPC

node both processes the received traffic and forwards it out. This way of working doubles the I/O traffic in the entire access path. This can be a bottleneck for performance improvement under high traffic loads.

- Data is not aware. The request to access storage in the computing cluster is forwarded to the back-end cluster through the gateway. Due to the load balancer, the destination of the request is not necessarily the location of the storage node. Therefore, the request needs to be forwarded again to reach the actual storage node. This forwarding not only increases the latency of completing a request by adding a network forwarding, but also increases the traffic load on each machine. The solution based on the network layer loses application information and fails to obtain the real location of storage nodes.

- Double the traffic. As we mentioned before, since the load balancer does not select the correct data node but only based on the load of each storage node, it may be necessary to forward the data one more time within the storage cluster to reach the correct storage node. This method of access causes each node to handle more traffic, both receiving a large number of data requests and forwarding the data requests to the actual storage nodes intact. Therefore, the effective bandwidth of each node is reduced by 1/2. This access method greatly increases the cost.

Another solution for the interaction between the computing cluster and the back-end storage cluster is to put the storage cluster and the computing cluster in the same VPC, as shown in figure 2. In this way, compute nodes can directly access nodes in the back-end storage cluster, so there is no need to forward through the gateway. This solution avoids the aforementioned doubling of traffic through gateways and the need for a high-performance, high-bandwidth gateway, but it raises new problems.

- A storage cluster in a VPC can only be accessed by the computing cluster in the same VPC, but cannot be shared by other computing clusters. To efficiently utilize resources, a storage cluster must be able to share resources with others. Therefore, multiple VPCs must be configured in a storage cluster to ensure that multiple computing clusters can access the storage cluster. The configuration of multiple VPCs is complicated, which also complicates subsequent operations. When something goes wrong, it can be very difficult to troubleshoot.

- Storage resources cannot be pooled. To efficiently use storage resources, a common practice is to pool resources in all storage clusters so that tenants can access the same storage resource pool. Permission control and QoS isolation are implemented by the upper-layer storage system. Pooling not only efficiently uses storage capacity, but also implements I/O service balancing. For example, some applications require high IOPS and some applications require large capacity. After storage pools are created, all applications use the same cluster to complement each other. However, if storage clusters and computing clusters share the same VPC, storage pools cannot be implemented because unlimited VPCS cannot be configured in a storage cluster to share resources with other clusters. Not being able to pool is not consistent with the current trend of resource unification.

3.3 Design Principles

Under the background of the separation of computing and storage in the past, the computing cluster mainly uses the two methods mentioned in Section 3.2 to access the storage cluster, which can no longer meet the requirements of lightweight and rapid application in the cloud native environment. To minimize overhead and maximize performance when computing clusters accessing storage clusters, PoV are designed with the following principles in mind:

Lightweight. PoV offloads a large amount of storage logic to MoC cards and back-end storage clusters, deploying only lightweight clients in the computing cluster. Applications in the computing cluster only need to call the encapsulated API to forward requests, without the need for protocol parsing on the computing side. All storage protocol-related data are parsed and encapsulated in the MoC card or the back-end storage cluster. Because the client has few functions, simple logic, and is encapsulated as an interface, there is little need to upgrade and the upgrade complexity is reduced. In addition, all traffic does not pass through the host's kernel stack, thus saving the host's CPU resources.

Software and hardware collaboration. PoV explores ways for software and hardware collaboration to maximize software efficiency. In order to avoid CPU resource contention between storage-related logic and computing logic, PoV uses the idea of hardware coordination to offload storage logic processing tasks to MoC and use dedicated hardware to process

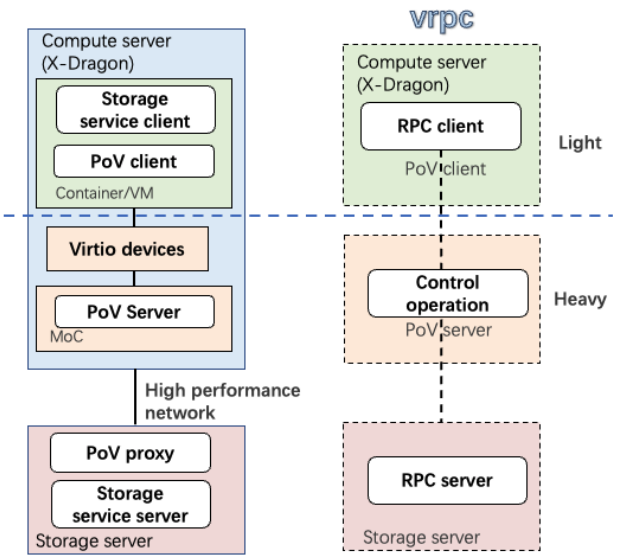

Figure 3: The overview of PoV

them. This combination of hardware and software can not only avoid the impact of computing services, but also speed up the processing of storage logic.

Security. PoV deploys only the interface programs used to invoke forwarding only to the computing side, so that the internal protocols of the back-end storage cluster are not exposed. Therefore, the program in the computing cluster cannot predict the storage logic of the back end to attack and so on, which ensures the security of user data.

4 Design

4.1 Overview

Pov is a virtual distributed storage service for cloud native applications' data access. In order to adapt the fast invocation and diverse cloud native applications, PoV separates its light part and heavy part during the data IO process and expose the light part to applications for fast storage service invocation (4.2). Benefits for the design of light-weight separation, PoV forms a fast path by offloading its heavy data IO path to hardware MoC to get high performance when accessing storage cluster for the cloud native applications, it also leverages high performance network protocol like RDMA to accelerate the data IO process (2.3). What's more, PoV naturally forms a isolation line that separates compute area and storage area, which effectively protect the storage area (4.4). Performance isolation and QoS mechanism do matter when providing storage service for multiple tenants with various demands, especially under the situation of Pangu as unified storage pool (4.5). Large scale online practice requires high availability design to guarantee the consistence of user's experience (4.6).

4.2 Light Weighted Storage Service

To realize fast storage service for quickly started cloud native applications. PoV separates the data IO path into light part and heavy part according to the complexity of processing data. As illustrated in Figure 4, PoV assigns a light weighted structure, namely storage service client, near the compute node so as to response application's storage invocation and read/write data to storage cluster. Thus, Pangu's storage service is abstracted as a light weight interface for cloud native applications.

The complex and heavy data processing and transmission process is finished by other components, which is transparent for cloud native applications. Besides, the loosely coupled components enable flexibly control operation on storage service. As a virtual storage data path, PoV can be abstracted as a VRPC (virtual remote procure call) service. The light part can be regarded as RPC client in compute node, the heavy part is processed by PoV server, then the storage service is done by RPC server located at the storage server. The separation makes PoV more open and high scalability, the control operation and other enhanced semantics can be implemented in PoV server.

The design of separating not only provides light weight service, but also realizes service-oriented goal. The standard light weighted interface can be service-oriented to respond to various applications' invocation. The heavy data control, data processing, and data transmission process is transparent for applications, which enables the service-oriented storage provision. For example, all applications can access Pangu through a light API $write()$ given by storage service client without paying any attention to the underlying implementation details.

4.3 High Performance Data Path

High performance storage service is highly demand for cloud native applications, for example, some short-lived and latency critical applications (e.g. online searching) are eager to get result from storage cluster, and some big data applications demand high throughput storage service. PoV builds a virtual distributed data path for applications to access Pangu's storage. In order to realize high performance data path, PoV has three main distinct innovations: (i) fast path by integrating hardware; (ii) routing in application layer; (iii) gathering data IO.

4.3.1 Low Latency via Hardware Integration

Benefits from the design of light-heavy separation, offloading the heavy part to MoC can be a solution to achieve low latency. MoC is versatile compute platform that sits in the data path and control path of applications. The unique location advantage makes MoC well-suited for the offload of complex storage access requests, including storage protocol, network protocol.

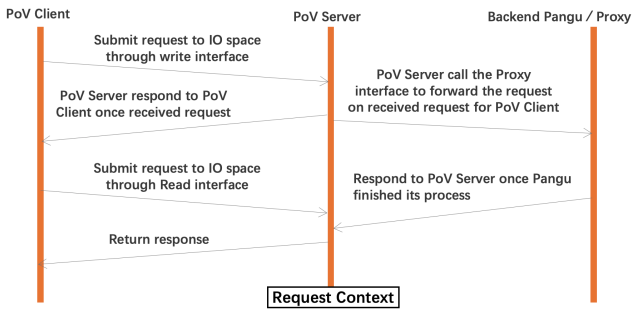

Figure 4: PoV's IO request.

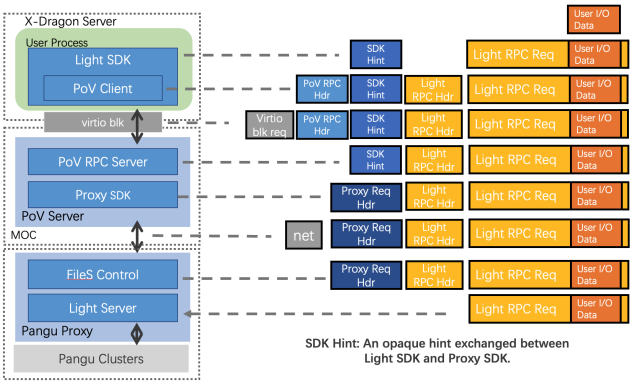

Figure 5: PoV's data path protocol.

Pangu has deployed large scale RDMA in its storage cluster to guarantee the performance of storage network [13]. In order to obtain low latency for cloud native applications to Pangu, PoV also leverages RDMA network protocol from compute node to storage cluster. Besides, it builds a fast path from cloud native applications and integrate this path with hardware MOC card.

Fast path. In particularly, PoV's data processing method consists control plane and data plane. Control plane is leveraged to configurate the mapping relationship of PoV's RDMA-based data queue and network connections. Data plane is mainly used to realize the data path, such as sending request and receiving response data. Control plane can establish a fast path for data path by the following steps. First, Application sends its own attribute information and file attribute information of the file to be accessed to PoV server. PoV server build data queue for the application once the application's authorization for the file is authenticated. Then the mapping relation between data queue and network connection is built, and a corresponding queue ID is configured for the data queue. Finally, the mapping relation is stored in PoV's mapping table.

The fast path have been built once the the mapping relationship of an application is established, the queue ID will be returned to application. Application can send request through data plane by carrying this queue ID. The request can be sent to back-end storage cluster through the fast path by searching the mapping table based on this queue ID. PoV response will be generated when application's request is finished by back-end storage cluster, and PoV response will be sent to application through the fast path.

4.3.2 High Throughput by Application layer routing

Also benefits from the decoupling of light part and heavy part in data path. The heavy part enables us to flexibly design control principles, such as application-oriented operations, including application layer's routing and application layer's blacking. Traditional network layer routing is not a viable option for locating the position of data because network layer fails to sense the position of storage position for given application. For example, the load balancer network will firstly send data request to a Panguproxy according to load balancing algorithm, and then the request be forwarded to the storage node that holds the data to be visited in the storage cluster. This kind of route mechanism makes request 2-hops of request in the Pangucluster, leading to the half of effective bandwidth in Pangu's storage cluster. However, hundreds of millions of concurrent requests requires high throughput, and we are motivated improve efficient bandwidth utilization.

Application layer's routing. To improve the throughput of Pangu's storage cluster, yet support the requests for large scale cloud native applications. PoV's data path is designed with application layer routing. Figure 5 shows PoV's data path and protocol processing. For a given IO data, Light SDK first generates a Light RPC Request and carrys a SDK hint. The SDK hint is mainly used to exchange information between Light SDK and Proxy SDK, including proxy location key, proxy rerouting instructions, and retry forbidden list. Proxy Req Header will encapulate SDK hint and other control information between Porxy SDK and PanguProxy, such file system information, trace switch, and load read information. The Proxy Req Header is leveraged to route the request according to a application layer route table. For similicity, we do not describe the protocol processing of virtio blk and network.

By the application layer's route method, a data IO from an application can sense the physical location of the storage data, only one hop network is needed from the application to the storage service. As a result, the network bandwidth and IOPS of Panguare greatly improved.

Gathering IO. To further improve the IOPS of Pangucluster for cloud native applications, PoV leverages gathering IO to reduce interrupt overhead. The IOPS is increased about 15% by gathering the IO requests.

4.4 Security

As we have illustrated in Figure 4. The back-end physical cluster is a secure area and physically isolated. The back-

end storage cluster's management and control method cannot be perceived by the front-end applications, such as the IP address of storage cluster, the internal data transfer protocol. Application's storage access request is transformed into a PoV request according to application's demands, and then this PoV request is transformed into a physical cluster-oriented real storage request. In this way, we realize a secure penetration from virtual domain to physical, which effectively protect the back-end storage cluster.

4.5 Performance Isolation under Shared Storage Pool

As a unified storage pool, Pangu serves requests from diverse tenants of different regions. For example, Pangu not only supports offline background writing IO but also supports online search writing IO. As a result, strong performance isolation method must be applied to guarantee the QoS of different tenants. As Figure 6 presents, the priority queue and scheduling strategy permeate every components in the data path from compute node to storage node. This full path QoS mechanism adequately guarantee that the tenants' different demands can be satisfied in a fine-grained way. For cloud native applications running in the front containers, the virtio-blk can be configured with bandwidth critical and latency critical to meet the different IO requests. In hardware MOC card, there is a dedicated and sophisticated scheduler to allocate resources to different IO requests, so as to process network protocol with different priority or with different resources. In storage layer, Motivated by the effective PID algorithm in the control system [14], we leverage PID algorithm during process of IO scheduling in the proxy of storage cluster to ensure performance isolation.

This full path QoS mechanism goes through physical devices like MoC, switch, and storage NIC, each of which is equipped with well-designed priority queue or scheduling strategy to isolate IO requests. And this QoS mechanism satisfies our goal that the data of online search business and offline batch tasks can both be stored in Pangu and improves the utilization of storage resource.

4.6 Availability

Availability is critical for providing stable storage service. Ensure the storage service quality of Pangu is challenging when the scale of storage cluster that leverage PoV as data path for large scale cloud native applications expands to a certain extent. In order to guarantee availability, many mechanisms are leveraged to solve the unexpected failover or crash problems. For example, the $keep-alive-mechanism$ is apllied to handle to failover of PoV's components. Specifically, after the PoV client creates the IO Request Queue, it periodically sends heartbeat messages to the PoV server through the admin queue.

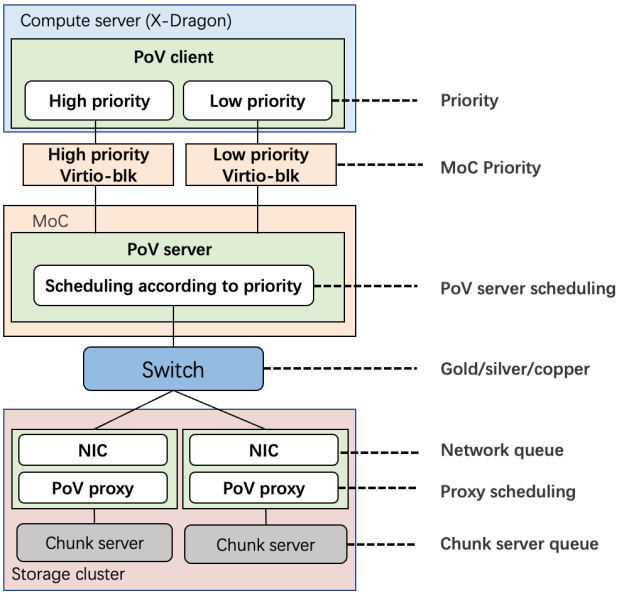

Figure 6: PoV's full path QoS.

LightSDK failover. Once the LightSDK failover, the heartbeat signal will lost, and the PoV server is triggered to recycle IO request queue resources. In the process of recycling, inflight IO will be processed. And the LightSDK will be restarted with a new process ID, thus it will apply for new IO request queue to visit Pangu.

To avoid wasting IO request queue resources when light SDK crashes, We apply the keep alive mechanism to collect the resources automatically. The resource collection process will be triggered if the PoV server does not receive the heartbeat message within a certain time. The resource collection process consists two steps: i) Request clean up. This process is done by PoV server without the participation of LightSDK. The inflight IO request will occupy some resource and these resource can not be released until the IO are returned by bacnkedn Pangu. In this step, the memory collection must synchronize with the hardware to prevent the problem of DMA polluting. ii) Resource collection. The created and allocated IO request queues are all released, their resources will be collected when all inflight requests have finished.

In some cases, LightSDK will send requests to IO request queue that had been reallocated to other SDK. Under this circumstance, PoV server is responsiable for check the request's information. Every request that is sent to PoV server contains process ID, and every IO request queue will hold a process ID field. PoV server will directly response the request and return an error code if the process ID in request mismatch with the process ID in the IO request queue.

PoV server failover. Once failover happens on PoV server, all LightSDK that related with fail to provide storage service for cloud native applications. Under this circumstance, we

have to deal with the returned data from Pangu and new requests from LightSDK. For returned data, all IO context information is lost when PoV server restarts, the returned data can not find its context, so we filter all returned data. For new requests, they will time out once PoV server failovers, and they will reprocessed all abnormal requests to ensure the correctness of IO requests.

By the design of $keep-alive-mechanism$ and other methods, PoV ensures the availability of cloud native applications' data access for Pangu .

5 Implementation

PoV's implementation in Pangu.

We implement PoV in x86 Linux hosts and ARM-based MoC. PoV is written in C++ with 2000 lines of code (LoC) for PoV server and 1000 LoC for PoV proxy.

PoV client runs on the client and implements the API interface layer provided to users. Users can access the back-end storage system through the API. At the same time, it encapsulates user requests on the basis of block devices and transmits data to PoV server in the MoC card in the form of RPC packets through virtio-blk.

PoV Server is deployed on the MoC of ARM architecture. It implements the SDK API for accessing back-end Proxy and the API for requesting RPC calls to Proxy. It forwards PoV requests sent from PoV client to back-end Proxy in the form of RPC.

PoV proxy resides in a storage cluster. It receives RPC requests from the PoV Server, processes request packets, and invokes back-end storage services according to the requests encapsulated by the PoV client.

In PoV, we abstract the concept of PoV device to solve the problem of communication between client and server. The access space for each PoV Device is divided into Device configuration space, Admin request space, and IO request space. The Device configuration space is a global read-only space, which is used to obtain PoV Device attributes. Based on the information, you can determine whether the Device is a valid PoV Device. The size of the space requested by Admin is less than 4KB, and the space is a global shared space. When multiple threads access Admin Queue, they need to use a shared file lock for concurrent mutual exclusion. Assign a separate access area to each SDK in the IO request space. Each SDK's access area consists of multiple IO Request queues. The size of an I/o request cannot exceed 1MB. Asynchronous I/OS access different I/o request queues to implement concurrent requests. In addition, space is reserved between each requested space to prevent Block layer from merging I/Os.

6 Evaluation

6.1 microbenchmark

Design experiments for the following goals: 1. performance. 2. availability. 3. scalability. 4. QoS.

6.2 Large scale experience

7 Related work

8 Conclusion

9 Footnotes, Verbatim, and Citations

Footnotes should be places after punctuation characters, without any spaces between said characters and footnotes, like so.[1] And some embedded literal code may look as follows.

```
int main(int argc, char *argv[])
{
    return 0;
}
```

Now we're going to cite somebody. Watch for the cite tag. Here it comes. Arpachi-Dusseau and Arpachi-Dusseau coauthored an excellent OS book, which is also really funny [10], and Waldspurger got into the SIGOPS hall-of-fame due to his seminal paper about resource management in the ESX hypervisor [26].

The tilde character (~) in the tex source means a non-breaking space. This way, your reference will always be attached to the word that preceded it, instead of going to the next line.

And the 'cite' package sorts your citations by their numerical order of the corresponding references at the end of the paper, ridding you from the need to notice that, e.g, "Waldspurger" appears after "Arpachi-Dusseau" when sorting references alphabetically [10, 26].

It'd be nice and thoughtful of you to include a suitable link in each and every bibtex entry that you use in your submission, to allow reviewers (and other readers) to easily get to the cited work, as is done in all entries found in the References section of this document.

Now we're going take a look at Section 10, but not before observing that refs to sections and citations and such are colored and clickable in the PDF because of the packages we've included.

10 Floating Figures and Lists

Here's a typical reference to a floating figure: Figure 7. Floats should usually be placed where latex wants then. Figure7 is

[1] Remember that USENIX format stopped using endnotes and is now using regular footnotes.

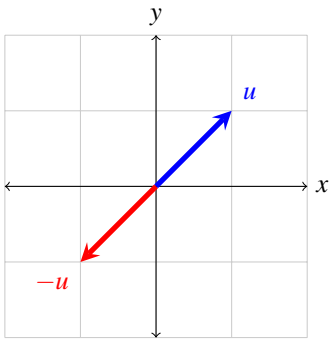

Figure 7: Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text. Text size inside figure should be as big as caption's text.

centered, and has a caption that instructs you to make sure that the size of the text within the figures that you use is as big as (or bigger than) the size of the text in the caption of the figures. Please do. Really.

In our case, we've explicitly drawn the figure inlined in latex, to allow this tex file to cleanly compile. But usually, your figures will reside in some file.pdf, and you'd include them in your document with, say, \includegraphics.

Lists are sometimes quite handy. If you want to itemize things, feel free:

fread a function that reads from a `stream` into the array `ptr` at most `nobj` objects of size `size`, returning returns the number of objects read.

Fred a person's name, e.g., there once was a dude named Fred who separated usenix.sty from this file to allow for easy inclusion.

The noindent at the start of this paragraph in its tex version makes it clear that it's a continuation of the preceding paragraph, as opposed to a new paragraph in its own right.

10.1 LaTeX-ing Your TeX File

People often use `pdflatex` these days for creating pdf-s from tex files via the shell. And `bibtex`, of course. Works for us.

Acknowledgments

The USENIX latex style is old and very tired, which is why there's no \acks command for you to use when acknowledging. Sorry.

Availability

USENIX program committees give extra points to submissions that are backed by artifacts that are publicly available. If you made your code or data available, it's worth mentioning this fact in a dedicated section.

References

[1] Alibaba cloud. Alibaba Cloud: Cloud Computing Services. https://www.alibabacloud.com/.

[2] Amazon. Amazon S3. https://aws.amazon.com/cn/s3/, 2021.

[3] Azure function queues. Azure Function Queues. https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-storage-queue, 2021.

[4] Cloud native. What is Cloud Native. https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition, 2021.

[5] Cloud native. Who we are. https://katacontainers.io/, 2021.

[6] Introducing nydus. Dragonfly Container Image Service. https://www.cncf.io/blog/2020/10/20/introducing-nydus-dragonfly-container-image-service/ 2021.

[7] Microsoft azure blob storage. Microsoft Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/, 2021.

[8] Microsoft azure traces. Azure Functions Traces. https://github.com/Azure/AzurePublicDataset, 2021.

[9] X-dragon. Operating Large Scale Kubernetes Clusters on Alibaba X-Dragon Servers. https://www.cncf.io/about/who-we-are/, 2021.

[10] Remzi H. Arpaci-Dusseau and Arpaci-Dusseau Andrea C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, LLC, 1.00 edition, 2015. http://pages.cs.wisc.edu/~remzi/OSTEP/.

[11] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2477–2489, 2021.

[12] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[13] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533. USENIX Association, April 2021.

[14] Jingqing Han. From pid to active disturbance rejection control. *IEEE transactions on Industrial Electronics*, 56(3):900–906, 2009.

[15] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[16] 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 *Proceedings of the 2021 International Conference on Management of Data*, pages 857–871, 2021.

[17] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 789–794, 2018.

[18] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 427–444, 2018.

[19] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21)*, pages 805–820, 2021.

[20] 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 *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 228–244, 2021.

[21] Alina Nesen and Bharat Bhargava. Towards situational awareness with multimodal streaming data fusion: serverless computing approach. In *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments*, pages 1–6, 2021.

[22] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 193–206, 2019.

[23] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. *arXiv preprint arXiv:2104.13869*, 2021.

[24] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 205–218, 2020.

[25] Akshitha Sriraman and Thomas F. Wenisch. µtune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, Carlsbad, CA, October 2018. USENIX Association.

[26] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 181–194, 2002. https://www.usenix.org/legacy/event/osdi02/tech/waldspurger/waldspurger.pdf.

[27] Piyush Yadav, Dhaval Salwala, Felipe Arruda Pontes, Praneet Dhingra, and Edward Curry. Query-driven video event processing for the internet of multimedia things. *Proceedings of the VLDB Endowment*, 14(12):2847–2850, 2021.

[28] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1187–1204, 2020.

[29] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.

[30] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161, 2018.