Chen Q, Chen K, Chen ZN *et al.* Lessons learned from optimizing the Sunway storage system for higher application I/O performance. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(1): 47–60 Jan. 2020. DOI 10.1007/s11390-020-9798-5

Lessons Learned from Optimizing the Sunway Storage System for Higher Application I/O Performance

Qi Chen[1], Kang Chen[1], Zuo-Ning Chen[2], *Fellow, CCF*, Wei Xue[1], Xu Ji[1,3], and Bin Yang[3,4]

[1]*Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, Beijing 100084, China*

[2]*Chinese Academy of Engineering, Beijing 100088, China*

[3]*National Supercomputing Center in Wuxi, Wuxi 214100, China*

[4]*School of Software, Shandong University, Jinan 250101, China*

E-mail: chenq17@mails.tsinghua.edu.cn; chenkang@tsinghua.edu.cn; chenzuoning@vip.163.com
xuewei@tsinghua.edu.cn; sov.matrixac@gmail.com; bin.yang@mail.sdu.edu.cn

Received July 30, 2019; revised November 28, 2019.

Abstract It is hard for applications to make full utilization of the peak bandwidth of the storage system in high-performance computers because of I/O interferences, storage resource misallocations and complex long I/O paths. We performed several studies to bridge this gap in the Sunway storage system, which serves the supercomputer Sunway TaihuLight. To locate these issues and connections between them, an end-to-end performance monitoring and diagnosis tool was developed to understand I/O behaviors of applications and the system. With the help of the tool, we were about to find out the root causes of such performance barriers at the I/O forwarding layer and the parallel file system layer. An application-aware I/O forwarding allocation framework was used to address the I/O interferences and resource misallocations at the I/O forwarding layer. A performance-aware data placement mechanism was proposed to mitigate the impact of I/O interferences and performance variations of storage devices in the PFS. Together, applications obtained much better I/O performance. During the process, we also proposed a lightweight storage stack to shorten the I/O path of applications with *N*-*N* I/O pattern. This paper summarizes these studies and presents the lessons learned from the process.

Keywords high performance computing, I/O interference, parallel file system, performance optimization, resource misallocation

1 Introduction

Sunway TaihuLight (TaihuLight)[1] was built to accelerate scientific research. It has 40 960 260-core SW26010 processors, with a peak performance of 125.43 FLOPS. The Sunway storage system provides global storage service for TaihuLight via the file system interface. It uses an I/O forwarding architecture[1–3] shown in Fig.1 (details in Section 2).

The Sunway storage system uses the Lustre file

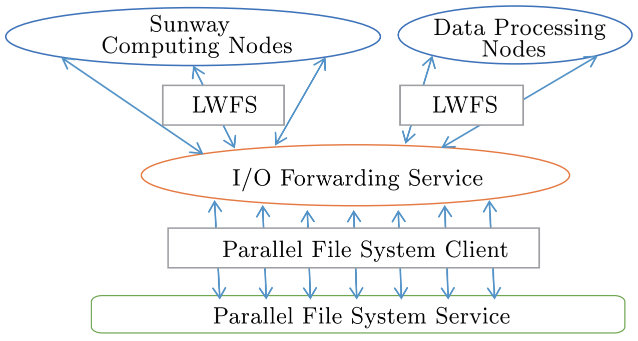


Fig.1. Sunway storage architecture.

Regular Paper

Special Section on Selected I/O Technologies for High-Performance Computing and Data Analytics

This work is supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000504, the Natural Science Foundation of China under Grant Nos. 61433008, 61373145, and 61572280, and China Postdoctoral Science Foundation under Grant No. 2018M630162.

[1]https://www.top500.org/system/178764, Nov. 2019.

system[4] as the backend parallel file system (PFS), which has an overall capacity of 10 PB and a peak bandwidth of 220 GB/s. Together with I/O forwarding services, the whole system supports over 40 000 computing nodes and data processing nodes. The storage system provides a POSIX compatible interface to ease the application development and to support legacy codes. During the daily use, the storage system supports the concurrent running of over 100 applications.

In such a complex system, applications can easily perceive low I/O performance compared with the peak performance of the PFS. For example, without tuning, ShenTu[5], a large-scale graph computing engine, obtained at most 20 GB/s throughput from one dedicated storage partition containing 140 storage nodes with an I/O bandwidth of 100 GB/s.

To find the root causes, a performance monitoring and diagnosis system, Beacon[6], was developed to understand I/O behaviors of applications and the system. With the help of this tool, we found that I/O interferences[7,8] and resource misallocations, which may happen at both the I/O forwarding layer and the backend PFS layer, hindered applications from getting high I/O performance from the storage system.

The I/O interferences came from the sharing nature of the I/O forwarding services and the backend PFS. I/O requests from different applications are received by the I/O forwarding services and then forwarded to the backend PFS. The requests are finally handled by the PFS. The competition for resources in the I/O forwarding layer and also the competition for storage devices in the PFS had led to I/O interferences and increased the request latencies. This made the I/O performance unbalanced. Some processes may lag behind other processes in the same application due to such imbalance. Because HPC applications usually run in a bulk synchronous parallel (BSP) model[9], the slowest process dragged down the overall progress of the application.

Resource misallocations have different reflections at different layers. Similar to other I/O forwarding systems, TaihuLight used a static I/O forwarding strategy, in which each I/O forwarding node served a fixed set of computing nodes. In such a configuration, larger-scale computing nodes had more I/O forwarding nodes. However, the requirement of computing resource and I/O resource in a job may not be proportional. The static I/O forwarding strategy easily made large-scale applications with less I/O requirement configured with a large number of I/O forwarding nodes while small-scale data-intensive applications configured with few ones. The misallocation hindered some applications from fully utilizing the performance of the backend PFS and while others wasted the forwarding resources[10]. Resource misallocations at the PFS layer were derived from performance anomalies and variances of storage devices[11,12]. Even if storage devices have had similar performance after their fresh installation, they behave differently after years of usage. However, most of the current parallel file systems seldom handle these anomalies and variations. The anomalies and variations made data distribution deviated from its original design assumption and easily led to performance imbalance[13,14].

To mitigate the impact of above issues on application I/O performance, we, cooperating with the storage system designers, developers and maintainers, performed some optimizations in the Sunway storage stack. Among them, the automatic application-ware I/O forwarding framework[10] was developed to address I/O interferences and resource misallocations at the forwarding layer. A performance-aware data placement strategy has been designed to mitigate the impact of I/O interferences and device performance variations in the PFS. It is worth mentioning that most of our optimizations were transparent to applications and application developers. During the process of optimizing the system, we also found some constraints of the PFS to support N-N I/O pattern. A project was started to overcome these constraints without breaking the POSIX interface of computing nodes.

Although the studies were performed in TaihuLight, these issues also exist in other HPC platforms[7,8,13,15,16]. Our experience is useful for those communities to design and implement an efficient storage stack to address these issues. This paper introduces our studies and presents lessons learned from the process.

The paper is organized as follows. The Sunway storage system architecture is discussed in Section 2 as well as the design considerations leading to this architecture. We discuss our end-to-end performance monitoring and diagnosis system in Section 3. Our solutions to bridge the performance gap and the lessons from them are presented in Section 4 (application-aware dynamic forwarding), Section 5 (performance-aware data placement), and Section 6 (remote node-local storage system) respectively followed by discussions in Section 7 and conclusions in Section 8.

2 Sunway Storage System Architecture

2.1 I/O Forwarding Design

Providing a POSIX-compatible interface is the design principle of the Sunway storage system. However, using Lustre native clients to implement this function in TaihuLight is not reasonable and impractical. The reasons are as follows.

Memory Consumption. In Lustre, the metadata server (MDS) and the object storage server (OSS) should cache the states of connected clients. Besides, the MDS also caches the objects that are opened on clients. With over 40 000 clients, MDS and OSS will consume a massive amount of memory. This increases the possibility of triggering out-of-memory failures (also reported by the spider storage system running 26 000 Lustre native clients[17]). Correspondingly, clients also keep the connection states to OSSes and MDS in memory. With 128 OSSes and 383 OSTs (object storage target), even without running any file system operation, each client requires over 500 MB of memory. Computing nodes in TaihuLight use many-core architecture (260 cores). Most of the memory should be reserved for applications and the relatively large memory consumption of Lustre clients is therefore not acceptable.

Server Overload. With too many clients accessing MDS concurrently, a large number of I/O requests will be queued. It can easily lead to network congestions and increase the possibility of request timeouts. This slows down the operations and even hangs the whole system. Another constraint comes from Lustre's distributed lock, which is the primary method to ensure global consistency. Frequent lock requests and revocations limit the performance of the system if clients share data, such as in the N-1 I/O pattern[18, 19].

Client Cache Paradox. Lustre heavily relies on client caches to improve I/O performance. However, file system cache in computing nodes brings few benefits in TaihuLight. HPC applications often read data once and cache it in the process virtual memory space, which makes the cache in filesystem client less effective. In addition, caching write data in computing nodes may violate the principle of checkpoint/restart semantics, in which, intermediate results are stored in the PFS for future job restarting in case of node failures. If data in the write cache of failed nodes do not have a chance to be flushed to the storage system, the checkpointing dataset will not be integrated and cannot be used by job restarting. Considering that TaihuLight has over 40 000 computing nodes, the mean time between failures (MTBF) of the computing system is relatively short. Caching the dirty data in computing nodes increases the risk of checkpoint data disintegration.

Based on the above considerations, the Sunway storage system abandoned the client cache and reduced the client number connecting to the PFS servers by using the I/O forwarding architecture, shown in Fig.1.

LWFS (lightweight file system) is used to perform the function of I/O forwarding. It consists of two parts, the LWFS-Client running on the computing node and the LWFSD running on the I/O forwarding node. LWFS-Client was implemented using the FUSE library[2] to keep POSIX compatibility. There is no data cache in LWFS-Client due to the reason discussed above. LWFSD talks to the Lustre file system using Lustre client. LWFSD caches metadata and data to find ways of optimization, such as combining small data. LDLM (Lustre Distributed Lock Manager) is used to guarantee cache consistency among LWFSD services. Each LWFSD uses a fixed number of workers (each CPU core runs one worker) to access the backend PFS to avoid overwhelming PFS with too many I/O requests.

Although no data caching in LWFS-client limited the performance of single computing nodes (the I/O bandwidth of one client is about 80 MB/s–100 MB/s), POSIX-compatible semantics alleviated much of the burden on application developers. Besides, the relatively low I/O performance of each client is not a problem because of the large number of computing nodes (over 40 000 nodes). The aggregated performance exceeds the total bandwidth of the PFS.

2.2 Consolidating I/O Forwarding Service with PFS Service

Another distinguishing and interesting design decision in the Sunway storage system is to run the I/O forwarding service (LWFSD) and the PFS service (OSS) on the same physical server. I/O forwarding services usually run on dedicated servers in HPC[1–3]. However, using dedicated servers for I/O forwarding nodes increases the number of servers and the scale of the storage network. We made the following observation[3] in TaihuLight: CPU is not the performance bottleneck

[2] https://github.com/libfuse/libfuse, Nov. 2019.

[3] In TaihuLight, each storage node has three LDISKFS based OSDs (object storage devices). The OSD uses disk array with hardware RAID. In this configuration, backend OSD has little pressure on CPU. However, in the configuration using ZFS based OSD with software RAID or having a large number of OSD on each node, this observation may be not applicable and need some evaluations.

for I/O forwarding and PFS service even when they run simultaneously on the same server.

The observation was verified by using a 10-node Lustre cluster comparing the results with separating and consolidating services. IOR (Interleaved or Random)[20], which is a widely used benchmark to evaluate the parallel I/O performance of a PFS, was used to generate the workload twice as the peak I/O that each forwarding service needs to serve in TaihuLight. Fig.2 shows the CPU load and device performance of a single storage node during the test.

The CPU is not fully utilized (about 15% idle) when consolidating the I/O forwarding service with the PFS service (Fig.2(b)). When they are separated, it is about 40% idle for CPU usage even with a heavy load (Fig.2(a)). In both cases, the disk performance of the server did not distinguish much (Fig.2(c) and Fig.2(d)). And the total bandwidth of the test cluster is also similar: 38 500.74 MB/s for write and 18 394.96 MB/s for read in the consolidated mode vs 36 775.26 MB/s for write and 19 237.42 MB/s for read in the separated mode. Thus, in Sunway storage system, we consolidated the I/O forwarding service with the PFS service. This saved the cost of the whole system, and also gave us some opportunities to shorten the I/O path of applications with *N*-*N* I/O pattern, which is presented in Section 6.

However, this consolidation introduced an unexpected problem of deadlock under heavy workloads. As shown in Fig.3, each server hosts one client and one

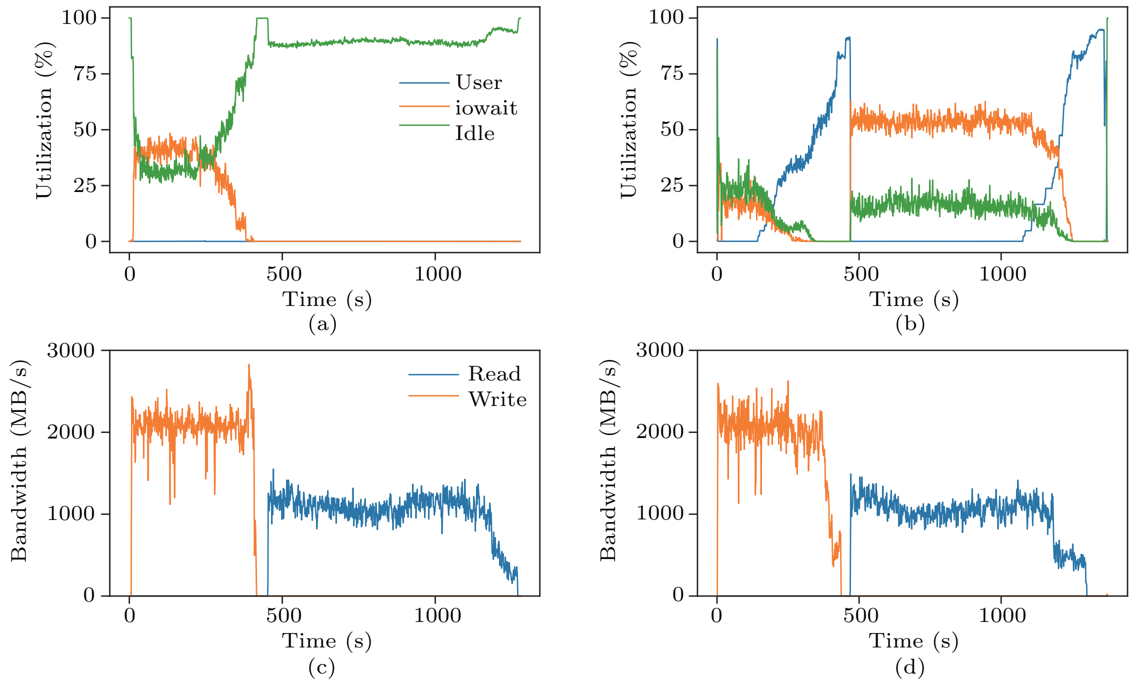


Fig.2. CPU load responding to high I/O workload on one storage node.

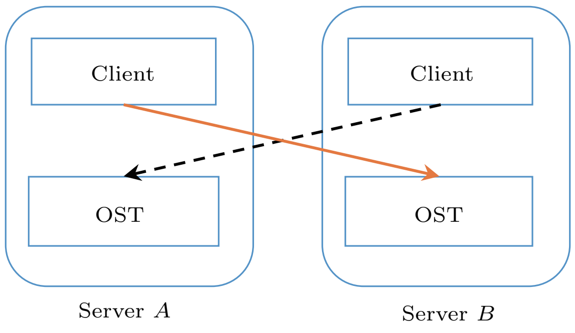


Fig.3. Deadlock between servers.

OSS. If both servers run out of their memory, each client needs to flush the data to the other OSS located in the remote server. Since OSS needs to allocate the memory to receive the new incoming data, the system gets stuck because there is no memory available. The whole PFS may hang in such a scenario. We solved this problem by memory isolation using the hypervisor KVM (Kernel Virtual Machine)[4]. OSS runs in the host operating system while I/O forwarding service (LWFSD and PFS client) runs in the guest operating system with

[4]https://www.linux-kvm.org/page/Main_Page, Nov. 2019.

pre-allocated memory pinned to the physical memory (never used by the host operating system). Thanks to the InfiniBand virtualization, the virtual I/O forwarding can provide I/O bandwidth of about 2.5 GB/s, which is almost the same as a physical I/O forwarding service can provide.

3 End-to-End Monitoring of Storage Performance Metrics

Finding out the connection between performance anomalies of the storage system and applications is the first step to optimize the system. Achieving this goal requires a comprehensive understanding of I/O behaviours of applications and the whole system. However, in a large-scale system with thousands of applications, it is not an easy task.

Applications running on HPC are from many different research areas and use various I/O libraries. It is impractical for the administrators to analyze I/O pattern of each application. Due to code complexity, coworking from different organizations, or the legacy code used, even the users may be not clear about their application I/O patterns. Thus, the storage system needs an automatic method to extract and detect I/O patterns of the applications running on it.

Learning about the I/O patterns of applications is not enough. The I/O interferences and resource misallocations, presented in Section 1, are important factors that limit application I/O performance. Without building the correlation of the full storage stack and the running jobs, these factors are difficult to dig out. Thus, an end-to-end performance analysis tool is also needed to bridge the understanding gap between the I/O behaviors of applications and the storage system.

Some performance monitoring and analyzing tools was implemented to help to understand the I/O behaviors [21, 22]. However, these tools can only solve one of the requirements that we mentioned above. To bridge the gap, the lightweight end-to-end I/O performance monitoring and diagnosis system Beacon [6] has been developed. Beacon performs a multiple level performance profiling. It transparently captures I/O workflows during job execution by instrumenting the I/O path between clients and servers in LWFS and collecting I/O performance metrics from the backend PFS. These data are compressed across multiple layers and then stored in a distributed database. Based on the collected data, Beacon can do end-to-end per-job I/O workflow analyses, I/O interference analyses among jobs and storage performance bottleneck findings. Since its deployment, Beacon has helped many developers to optimize their application I/O patterns. It also found many system-level performance issues, such as PFS client cache thrashing, performance anomalies and I/O interferences at both the I/O forwarding layer and the PFS layer. Beacon gave us the insightful knowledge to optimize the storage system.

During the development of Beacon, collection and correlation of a large volume of data are challenging, which are summarized as follows.

Lesson 1. Controlling the data volume is the key to a successful multiple level performance profiling system. Small data volume makes the analysis inaccurate, but large volume impairs application I/O performance. Data deduplication and compression are helpful for this issue. In Beacon, similar data items in each layer are deduplicated, and data across different layers are compressed to reduce the total size.

Lesson 2. The POSIX interface allows little information of applications passed into the storage stack. Without this information, it is difficult to correlate the performance data of one job across multiple layers when many jobs concurrently run in the system. Thus, storage systems should provide some mechanisms to bridge the gap. We are considering extending POSIX API to pass application hints (such as job identifier) to the storage system and tag them in every I/O request to ease the classification of I/O requests.

4 Application-Aware Dynamic I/O Forwarding

The I/O forwarding nodes are gateways to the backend PFS. The number and the performance of I/O forwarding nodes that a job uses limit the overall performance it can get from the backend PFS [2]. TaihuLight originally used a static I/O forwarding, in which a forwarding node serves a fixed set of computing nodes. This method has several problems.

Firstly, I/O requests from different jobs easily collided on the I/O forwarding nodes. The I/O interferences reduced the performance of the forwarding nodes. This was mainly caused by the threading model of LWFSD. LWFSD used a fixed number of worker threads to handle I/O requests and gave metadata operations a higher priority. This model helped to reduce the pressure to the PFS and the latency of metadata operations. But it also had some side effects: high priority requests (metadata operations) starved the low priority requests (read/write operations); high latency requests (requests with larger and slower I/O) may block all the other requests. In the actual use, the slowdown can reach about 10 times [10].

Secondly, in the static mapping, application I/O requirement cannot be always matched with the capacity of the forwarding resources, and as a result, applications cannot make full utilization of the PFS performance. After investigating some outstanding applications, we found that some jobs running on a large number of computing nodes with many I/O forwarding nodes serving them used 1-1 I/O pattern to write to or read from the storage system, while the I/O performance of many small-scale I/O-intensive jobs with N-N I/O pattern was limited by fewer I/O forwarding nodes [6].

Aware of these limitations, we divided the I/O forwarding nodes into two groups. The first group contained about 80 nodes used as static I/O forwarding nodes. Each I/O forwarding node serves 512 computing nodes. The other group contained about 200 nodes, which were reserved for I/O-intensive jobs. To use the reserved I/O forwarding resources, users propose I/O requirements to the administrators, and, if approved, the corresponding resources are allocated from the reserved group.

However, such seemingly reasonable principle worked unexpectedly in practice. The reserved 200 forwarding nodes were rarely used, and most applications still used the default mapping of I/O forwarding nodes. The reason was that most of the users cannot estimate how many I/O forwarding resources they should apply. Or even, they did not put much attention to the storage system usage until the I/O performance severely constrained the runtime of the applications.

To address the above problems, the application-aware dynamic forwarding resource allocation (DFRA) [10] was developed. The mechanism automatically evaluates the I/O requirements of one job when restarted and allocates corresponding I/O forwarding resources to avoid under- or over-utilization of I/O forwarding resources and the I/O interferences from other jobs. To achieve the goal, it first extracts the I/O patterns and performance metrics from historical runs using Beacon mentioned in Section 3. Then, it calculates the requirement of the forwarding resources and generates resource allocation hints on the current system usage. At last, the hints are passed to the job scheduler to do the remapping. The above process is transparent to applications and users. After being deployed in TaihuLight, it improved the I/O performance of some applications by up to 18.9x and saved over 200 million core-hours in 11 months.

Two lessons were learned from this optimization process that, we believe, are useful for the design and implementation of the I/O forwarding architecture.

Lesson 3. The static I/O forwarding strategy is simple to deploy and very efficient when the I/O load is balanced. But it is vulnerable to I/O interferences and inefficient I/O patterns, such as N-1 and 1-1 I/O pattern. Modern HPC system always concurrently runs many jobs with different I/O patterns. Thus, it is desired to develop to an automatic and dynamic I/O forwarding resource allocation mechanism to match the true I/O requirement with the I/O forwarding resources. And then, end-to-end performance monitoring and analysis tools, such as Beacon, are required and helpful.

Lesson 4. Interferences from different types, sizes and priorities of I/O requests decrease the total performance of the I/O forwarding node. I/O forwarding mechanism should adopt efficient methods to mitigate this problem. For this purpose, the system can avoid different jobs using the same I/O forwarding nodes, just like we did in DFRA. Besides, I/O handling routine can be elaborately designed to avoid I/O interferences. For example, worker threads in the LWFSD using synchronous mode to process I/O requests easily make long latency I/O requests block other requests and starve the backend PFS. To mitigate this problem, the I/O handling routine can be divided into more stages and reconstructed with some asynchronous mechanisms.

5 Performance-Aware Data Placement

I/O interferences and resource misallocations at the PFS layer are also the critical factors that limit the application I/O performance.

The I/O interferences are mainly from different I/O patterns of concurrently running jobs. Fig.4 illustrates the I/O interference between two concurrent jobs.

Job A uses N-N I/O pattern, and Job B uses 1-1 I/O pattern. The storage system has only 4 OSTs, and the round-robin algorithm is used to place objects. Job A is first started and its data are distributed on the four nodes, and then B is started after that. In this case, A and B compete for storage resources on OST00. Each gets half of the performance. Although the processes of A on OST01-03 are completed earlier, it has to wait for the process on OST00 to be completed. This interference is very common. The statistics of I/O patterns of applications running in TaihuLight are listed in Table 1 [6].

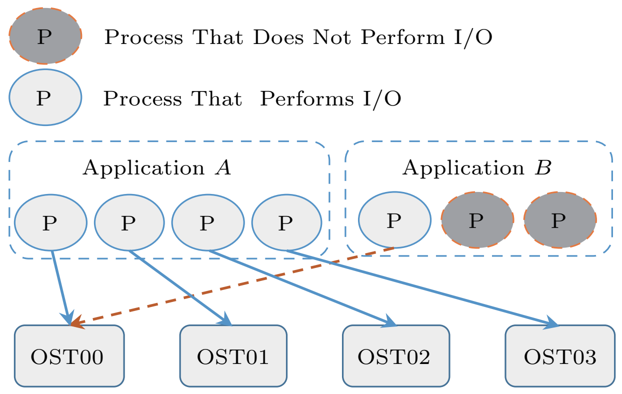


Fig.4. Interference of two parallel jobs.

Table 1. I/O Pattern Statistics in TaihuLight

I/O Pattern	Avg. Read Volume (GB)	Avg. Write Volume (GB)	Job Count
N-*N*	96.8	120.1	11 073
N-*M*	36.2	63.2	324
N-1	19.6	19.3	2 382
1-1	33.0	142.3	16 251

From the table, we can see that more than a half of jobs run in *N*-1 and 1-1 I/O pattern. When co-running with jobs using *N*-*N* I/O pattern, the I/O interferences mentioned above are unavoidable under the round-robin allocation.

This problem can be mitigated by well arranging file layouts of applications. For example, if file layouts in Fig.4 are arranged as in Fig.5, the total I/O time of *A* is kept the same but *B* can run twice faster. Besides, one OST node is even free for serving other jobs.

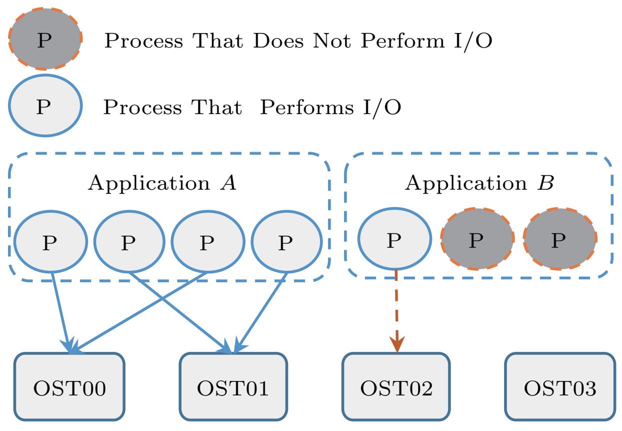


Fig.5. Better resource allocation mitigating interferences.

The resource misallocations are derived from performance anomalies and variances of storage devices. Although homogenous devices are used to build a PFS, the performance of the devices may differ after the storage system runs for a long time because of device aging[23], file system fragmentation[12] and disk array degradation. However, the popular data placement algorithms, such as the round-robin algorithm or hashing[24], cannot handle these anomalies and variances, and easily make file objects misallocated on storage nodes.

One of our maintained parallel file systems (based on lustre-1.8.1) encountered such a performance problem. This system has 240 OSSes, each equipped with one OST. Before putting into production use, the performance of each OST was about 800 MB/s, and the overall bandwidth of the system was more than 180 GB/s. The system had been carrying I/O-intensive applications for about four to five years. A large number of disks and array controllers had been replaced due to failures. During the use of the system, users always complained that their application I/O performance was very low. They even tried to spread their data across all of the servers. But this approach inversely reduced the performance.

To find the cause for this anomaly, we re-evaluated the performance of the OSTs. The result is shown in Fig.6.

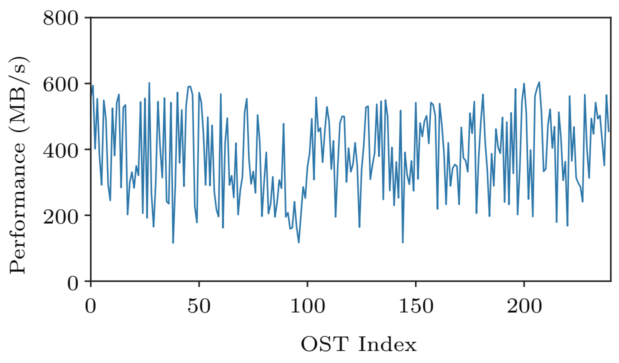


Fig.6. OST performance variances.

After a long period of production use, the performance of OSTs showed significant variances. The difference between the maximum performance and the minimum performance had been about 5 times. The files located on the slowest OST slowed down the entire I/O process. Aware of this problem, we added OSTs with similar performance into one pool, and put files from the same parallel I/O into the pool. As a result, the application I/O performance was improved by several times.

Similar things happened to the application ShenTu[5], which read 136.9 TB of graph data from the

PFS using N-N I/O pattern ($N = 10\,240$). As mentioned above, the application got 20 GB/s bandwidth at most before performing the optimization. Beacon found that some of the storage nodes involved had significant performance anomalies when performing I/O. When the data was migrated from these anomalous nodes, the I/O bandwidth of the application increased to about 70 GB/s.

We believed that efficiently controlling file layouts of different jobs is the key to mitigate the impact of the I/O interferences and resource misallocations in PFS. A performance-aware data placement framework was proposed to ease the process.

The I/O requirement of an application is described using an abstract of a resource pool. One resource pool consists of multiple OSTs, each of which has a variable *max_perf* to denote the maximum performance that the PFS distributes to it. If one OST is shared by multiple pools, some QoS mechanisms, such as TBF (token bucket filter)[25], are used to isolate the operations (including reads and writes) from different pools and guarantee each pool getting corresponding performance from the OST. One resource pool can only be used by one running job simultaneously, and the sum of *max_perf* of OSTs in the pool is the maximum bandwidth the job can get.

To ease the resource pool allocation and to address the performance variances of storage devices, OSTs in one resource pool are allowed to have different *max_perf*. This requires a data placement algorithm to distribute files from parallel I/O processes according to the *max_perf* of OSTs in a resource pool. Some QoS data placement algorithms, such as weighted CRUSH[26], and weighted random allocations[5][6], are probabilistic algorithms. When used to distribute files from parallel I/O processes of a job, they cannot assure that the data distribution is strictly proportional to the performance of storage devices, especially when the scales of files are small. This, as a result, may worsen the performance of parallel I/O. Thus, we proposed a deterministic data placement algorithm. It records the load of each OST in the resource pool and selects OSTs having the lightest load to create files. Because one resource pool is only used by one running job, the concurrent I/O requests in a resource pool mainly come from parallel I/O processes of a job. This simplifies the quantization of the storage load a lot. Here we use the number of opened objects on one OST, represented as *open_count*, to record the load of that node and a ratio of *max_perf* and *open_count* + 1 to weigh the performance that new created file may get[7]. For each file creation, the algorithm puts the object (or stripe) on the node that has maximum evaluated weight.

We implemented the algorithm in Lustre-2.10 and built a Lustre file system with 8 OSSes, each equipped with one OST, to evaluate the mechanism. We created a resource pool. The performance of OSTs in the pool is shown in Table 2. The total bandwidth of the resource pool is 2 600 MB/s. TBF[25] is used to guarantee each OST getting the corresponding *max_perf*.

Table 2. Initial Performance of Each OST in the Test

OST Index	*max_perf* (MB/s)
1	500
2	300
3	500
4	200
5	300
6	100
7	500
8	200

Firstly, we compared our algorithm with the round-robin algorithm. In the test, we ran IOR with N-N I/O pattern on 20 Lustre clients. Each file has one object (or stripe). Each client ran 8 processes and each process wrote 1 GB data to the file system with a 1 MB transfer size. We also hacked the IOR code to record the I/O time of each parallel I/O process to get the I/O time of the slowest process on each OST. The object distribution and the I/O time of the slowest process are shown in Fig.7 and Fig.8 respectively.

From the results, we can see that the round-robin algorithm distributes the objects evenly across the OSTs, but the I/O time of processes on different OSTs is unbalanced because of the performance variations of OSTs. While our algorithm distributes the objects according to the *max_perf* of OSTs, and gets balanced I/O time of each parallel I/O process.

[5]https://jira.whamcloud.com/browse/LU-9, Nov. 2019.

[6]https://jira.whamcloud.com/browse/LU-9809, Nov. 2019.

[7]When a storage device is overloaded by many concurrent threads, the total performance of it will decrease. We use TBF[25] to provide performance guarantee of one OST. This algorithm can avoid concurrent threads overloading the storage device by limiting the requests processing rate. For systems that cannot avoid concurrent threads overloading the storage device, a penalty may be added to this weight.

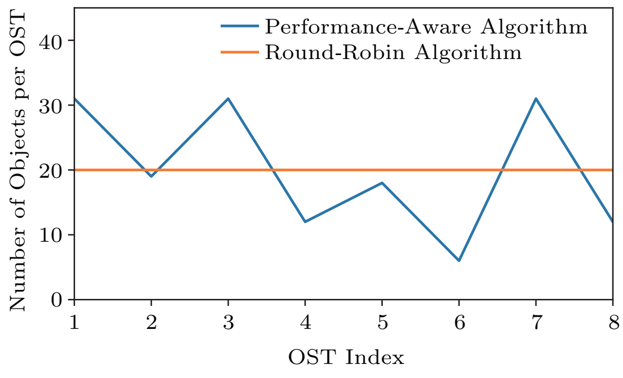


Fig.7. Object distribution on each OST.

Next, to show the effectiveness of the algorithm at different scales of parallel I/O, we tested the algorithm using different N values (100, 200, 400, 600, 800, 1 000). We used the same configuration in the previous test, except that each Lustre client ran 10 IOR processes. Totally, we used 100 Lustre clients. The object distribution and the I/O time of the slowest process on each OST are shown in Fig.9.

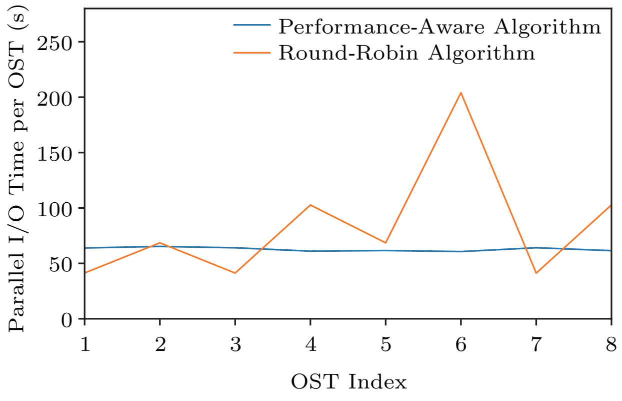


Fig.8. I/O time of the slowest process on each OST.

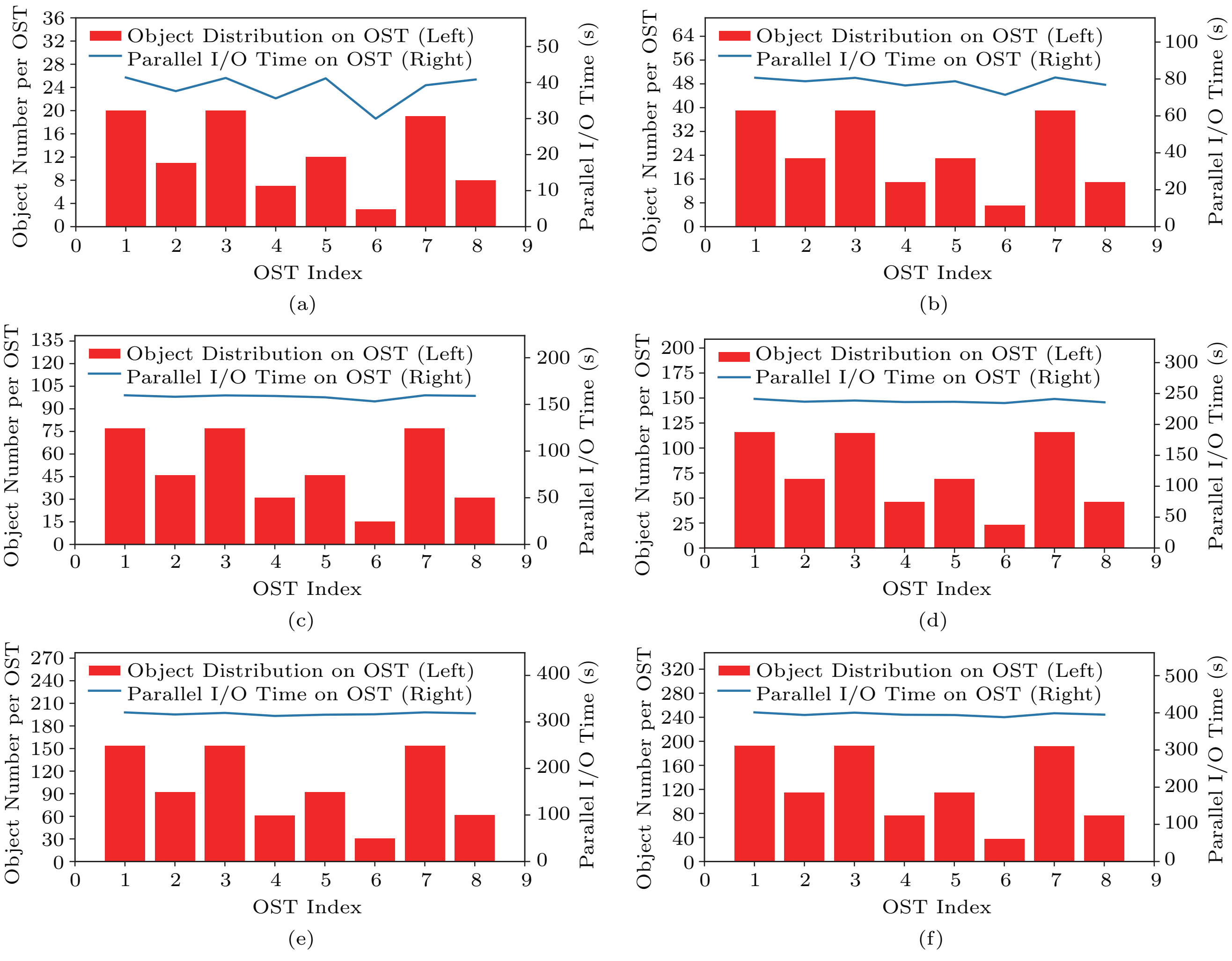


Fig.9. Object placement and parallel I/O time on each OST. (a) $N = 100$. (b) $N = 200$. (c) $N = 400$. (d) $N = 600$. (e) $N = 800$. (f) $N = 1000$.

The object distribution of the parallel I/O is proportional to the *max_perf* of each OST. Accordingly, the average I/O performance of each parallel I/O process is similar on the OSTs, which, as a result, reduced the total application parallel I/O time. We also see that when $N = 100$, the I/O time on OST 6 is about 10 seconds less than that on the slowest OST. That is because 100 objects cannot be strictly distributed to OSTs with no remainder, and as a result, the I/O load on OST 6 is slightly lighter than that on OST 1, 3, 5, 8.

We recorded the total bandwidth of IOR in each test in Table 3. The application almost got the maximum bandwidth of the resource pool.

Table 3. Total Bandwidth of IOR with Different N Values

N	Bandwidth (MB/s)
100	2 473
200	2 534
400	2 563
600	2 545
800	2 555
1 000	2 552

Lesson 5. Performance should be taken as a quantitative factor similar to the disk space or the number of servers in the PFS design. When allocating storage resources, the system can limit the amount of disk space usage or the number of data servers that files can be distributed to. However, most systems do not provide performance guarantee. This makes the PFS unable to cope with I/O interferences and performance variances. To address this problem, the resource pool with performance-ware data placement algorithm, mentioned above, is a feasible method. The abstract of the resource pool can easily be integrated with literary QoS methods. The algorithm can balance the performance of parallel I/O from one job even at small scale.

6 Remote Node-Local Storage System

In HPC, the N-N I/O pattern is usually used to improve application I/O performance[18, 27]. It is also the recommended method for write-intensive applications in TaihuLight. In this pattern, each process of the running job periodically flushes its data to the storage system in one or several directories using private file mode. The writing of each process can be bandwidth-dominated, in which each process stages out large chunks of data (several MB or GB) to the storage system, or IOPS (input/output operations per second) dominated, in which each process frequently opens the file, writes small size of data (several B or KB) and finally closes it.

In the I/O forwarding architecture similar to the Sunway storage system, Lustre may hinder the bare performance of storage devices. The reasons are as follows.

Metadata Bottleneck. Lustre uses a centralized metadata server to handle metadata operations. The metadata server in Lustre-2.5, which is used in TaihuLight, provided about 15k IOPS. However, for N-N I/O pattern, there are often tens of thousands and even hundreds of thousands of open or create operations bursting into the metadata server. The limited metadata performance became the bottleneck, especially when the application is IOPS-dominated. Besides, only one or several shared directories are used in N-N I/O pattern. Parallel file creations in shared directories from many clients incur high overhead on metadata consistency and get poor performance[28, 29].

Uncontrolled I/O Path. In the Sunway storage system, the I/O forwarding service and the OSS run on the same server. However, the I/O requests of N-N I/O may go through two distinct I/O paths illustrated in Fig.10. It is evident that Fig.10(a) has a longer I/O path, which increases the I/O latency and wastes the network bandwidth. In Lustre, locations of files are transparent to the users, and thus, the I/O path of 10(a) is inevitable. Besides, even when some methods are used to constrain I/O path to 10(b), the decoupled I/O stack of the client and OSS also brings about some overhead.

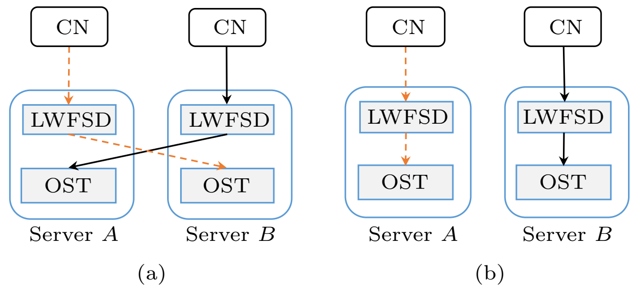


Fig.10. I/O paths when supporting N-to-N I/O pattern.

A remote node-local storage architecture is proposed to break these limitations. The idea is similar to some node-local burst buffer system, which provides a private storage space for each computing node. The difference is that we do not deploy storage devices in computing nodes, but export remote dedicated directories on the storage nodes to computing nodes using

LWFS. The exported directories are only used by one computing node. That is why we called it the remote node-local storage system.

Another difference with the prior work [30–32] is that we provided two views to address metadata bottleneck for running jobs, as shown in Fig.11.

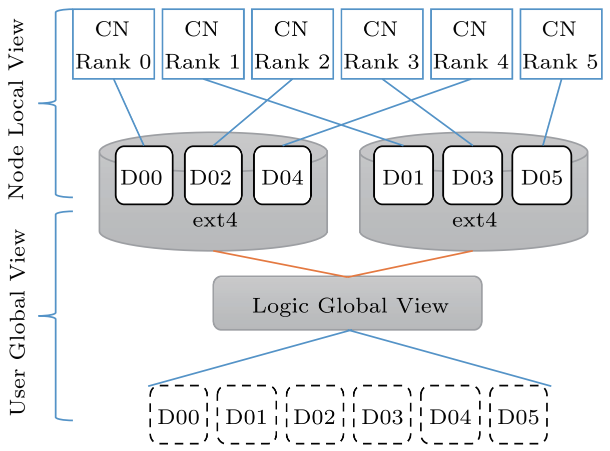


Fig.11. Remote node-local storage architecture.

Node-Local View for Parallel I/O Processes. The storage space of each storage node is divided into many directories. The directories are exported by LWFSD to the computing nodes. During the application running, each computing node only accesses its remote private directory, just like virtual disk access from the remote guest OS. By this way, our method not only shortens I/O path of N-N I/O processes but also eliminates the global metadata operations during the job execution. This also implies that our proposal only works for applications with N-N I/O pattern. When a job restarted, the computing resources assigned to one job may be changed. To make the node-local view valid to each process after the job is restarted, the mapping between the processes and the private directories should be unchanged. In TaihuLight, the job scheduler assigns a unique *rank-id* to each process in a job. Using this feature, a hash of *rank-id* is used to do the mapping. To reduce the overhead of re-attaching the remote storage space to the computing nodes (CN) when restarting because of node failures, we assigned the *rank-ids* of the failed nodes to the new replaced nodes and kept the others' *rank-id* unchanged, as illustrated in Fig.12. By this way, only the replaced nodes have to do the mapping.

Logic Global View for Data Management. Users sometimes need to check the status of the data their job generated or copy the results back to the PFS for other usages. Therefore, a global view of the separate datasets is also needed. However, implementing this function is much easier than that in PFS because a read-only view with eventual consistency guarantee is enough. In this project, we implemented a combining mechanism to aggregate the metadata information of one job from all of the I/O forwarding nodes. To accelerate the process, a distributed in-memory database was used to index the metadata information on the I/O forwarding nodes. Because the load of the interactive operations from users is much lighter than that from running jobs, this simple method was enough.

Using this method, the processes of N-N I/O pattern got comparable performance with using PFS, but without introducing any network traffic among storage nodes. Moreover, in favor of the separated namespace of each computing node, the metadata performance of applications with N-N I/O pattern was greatly improved.

Lesson 6. Not all scientific applications require the storage system to provide a global strong consistent namespace at all the time. Making some parallel I/O operations get rid of the constraint of strong consistency semantics improves the I/O performance and simplifies the design and implementation of the storage sys-

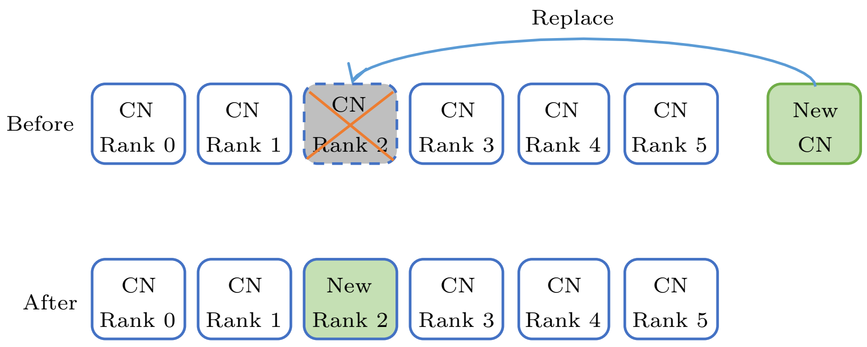


Fig.12. Remapping when replacing failed CN.

tem. The storage developers and designers should deliberately understand the requirements of each stage of application I/O, adopt appropriate consistency model, and build appropriate views (global or private) for applications to avoid unnecessary overheads.

7 Discussion

Applicability to Other Supercomputer Systems. The I/O forwarding architecture is widely used in HPC[1, 33]. The optimizations presented in this paper are also applicable to other I/O forwarding systems. The Sunway storage system was built upon the Lustre filesystem. We made some special designs to break some constraints of Lustre, such as virtual I/O forwarding nodes in Subsection 2.2 and remote node-local system in Section 6. These designs are helpful for other communities that use the Lustre file system. Except that, the optimizations mentioned in this paper are not Lustre specific. Some popular parallel file systems[34] also lack efficient mechanisms to handle the I/O interferences and resource misallocations. The resource pool and the application-aware data placement provide some references to address these problems in those systems. Besides, the architecture of the remote node-local storage system is similar to the shared burst buffer system. The idea may be helpful for the design of shared burst buffer file systems.

Implications for Future Supercomputer Designs. I/O performance is becoming more important for HPC platforms because modern applications, such as high-resolution simulation, graph computing and deep learning, access or produce huge volumes of data. Another trend is that applications with different I/O patterns concurrently running in a supercomputer system are becoming common. I/O interferences and resource misallocations will constrain application I/O performance and reduce the efficiency of the whole system. I/O performance isolation or, even QoS control in the storage system becomes a must. However, these features are missing in the HPC storage infrastructures. There are some excellent studies[35–37] to mitigate interferences among different workflow executions in the data processing systems. These studies give inspirations to solve the issues in HPC. But these technologies cannot be directly applied to HPC systems. For one thing, I/O requests have longer I/O path in HPC, and QoS control in one component is not enough; for another, high-performance systems should not only avoid I/O interferences among jobs but also keep the I/O performance of parallel I/O processes inside a job balanced. Our optimizations are performed in the whole storage stack. They provide some tips and references for this scenario.

8 Conclusions

In HPC, many applications concurrently accessing the shared storage system will incur I/O interferences among applications and resource misallocations in the storage stack. These factors prevent the applications from making full utilization of the storage I/O bandwidth. We did an end-to-end I/O analyzing of the applications and the whole storage stack, and made some optimizations at the I/O forwarding layer and the PFS layer to improve application I/O performance. We also proposed a lightweight storage stack to shorten the I/O path of applications with N-N I/O pattern. This paper presents these studies and lessons learned from them.

In our current work, we are focusing on improving application I/O performance. In future work, we plan to extend our studies to offer some I/O performance guarantee for HPC applications.

References

[1] Vishwanath V, Hereld M, Iskra K, Kimpe D, Morozov V, Papka M E, Ross R, Yoshii K. Accelerating I/O forwarding in IBM Blue Gene/P systems. In *Proc. the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2010, Article No. 34.

[2] Ohta K, Kimpe D, Cope J, Iskra K, Ross R, Ishikawa Y. Optimization techniques at the I/O forwarding layer. In *Proc. the 2010 IEEE International Conference on Cluster Computing*, September 2010, pp.312-321.

[3] Ali N, Carns P, Iskra K, Kimpe D, Lang S, Latham R, Ross R, Ward L, Sadayappan P. Scalable I/O forwarding framework for high-performance computing systems. In *Proc. the 2009 IEEE International Conference on Cluster Computing and Workshops*, August 2009, Article No. 10.

[4] Schwan P. Lustre: Building a file system for 1000-node clusters. In *Proc. the 2003 Linux Symposium*, July 2003, pp.380-386.

[5] Lin H, Zhu X, Yu B, Tang X, Xue W, Chen W, Zhang L, Hoefler T, Ma X, Liu X. ShenTu: Processing multi-trillion edge graphs on millions of cores in seconds. In *Proc. the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2018, Article No. 56.

[6] Yang B, Ji X, Ma X *et al*. End-to-end I/O monitoring on a leading supercomputer. In *Proc. the 16th USENIX Symposium on Networked Systems Design and Implementation*, February 2019, pp.379-394.

[7] Yildiz O, Dorier M, Ibrahim S, Ross R, Antoniu G. On the root causes of cross-application I/O interference in HPC

storage systems. In *Proc. the 2016 IEEE International Parallel and Distributed Processing Symposium*, May 2016, pp.750-759.

[8] Gainaru A, Aupy G, Benoit A, Cappello F, Robert Y, Snir M. Scheduling the I/O of HPC applications under congestion. In *Proc. the 2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp.1013-1022.

[9] Valiant L G. A bridging model for parallel computation. *Communications of the ACM*, 1990, 33(8): 103-111.

[10] Ji X, Yang B, Zhang T, Ma X, Zhu X, Wang X, El-Sayed N, Zhai J, Liu W, Xue W. Automatic, application-aware I/O forwarding resource allocation. In *Proc. the 17th USENIX Conference on File and Storage Technologies*, February 2019, pp.265-279.

[11] Gunawi H S, Suminto R O, Sears R *et al*. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage*, 2018, 14(3): Article No. 23.

[12] Djordjevic B, Timcenko V. Ext4 file system performance analysis in Linux environment. In *Proc. the 11th WSEAS International Conference on Applied Informatics and Communications*, August 2011, pp.288-293.

[13] Lofstead J, Zheng F, Liu Q, Klasky S, Oldfield R, Kordenbrock T, Schwan K, Wolf M. Managing variability in the IO performance of petascale storage systems. In *Proc. the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2010, Article No. 35.

[14] Kim Y, Gunasekaran R. Understanding I/O workload characteristics of a Peta-scale storage system. *The Journal of Supercomputing*, 2015, 71(3): 761-780.

[15] Dillow D A, Shipman G M, Oral S, Zhang Z, Kim Y. Enhancing I/O throughput via efficient routing and placement for large-scale parallel file systems. In *Proc. the 30th IEEE International Performance Computing and Communications Conference*, November 2011, Article No. 6.

[16] Lockwood G K, Snyder S, Wang T, Byna S, Carns P, Wright N J. A year in the life of a parallel file system. In *Proc. the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2018, Article No. 74.

[17] Shipman G, Dillow D, Oral S, Wang F, Fuller D, Hill J, Zhang Z. Lessons learned in deploying the world's largest scale Lustre file system. In *Proc. the 2010 Cray User Group Conference*, May 2010.

[18] Bent J, Gibson G, Grider G, McClelland B, Nowoczynski P, Nunez J, Polte M, Wingate M. PLFS: A checkpoint filesystem for parallel applications. In *Proc. the 2009 Conference on High Performance Computing Networking, Storage and Analysis*, November 2009, Article No. 26.

[19] Liao W K, Choudhary A. Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In *Proc. the 2008 ACM/IEEE Conference on Supercomputing*, November 2008, Article No. 3.

[20] Shan H, Antypas K, Shalf J. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proc. the 2008 ACM/IEEE Conference on Supercomputing*, November 2008, Article No. 42.

[21] Liu Y, Gunasekaran R, Ma X, Vazhkudai S S. Automatic identification of application I/O signatures from noisy server-side traces. In *Proc. the 12th USENIX Conference on File and Storage Technologies*, February 2014, pp.213-228.

[22] Carns P H, Latham R, Ross R B, Iskra K, Lang S, Riley K. 24/7 characterization of petascale I/O workloads. In *Proc. the International Conference on Cluster Computing*, August 2009, Article No. 75.

[23] Conway A, Bakshi A, Jiao Y, Jannen W, Zhan Y, Yuan J, Bender M A, Johnson R, Kuszmaul B C, Porter D E, Farach-Colton M. File systems fated for senescence? Nonsense, says science! In *Proc. the 15th USENIX Conference on File and Storage Technologies*, February 2017, pp.45-58.

[24] Awerbuch B, Scheideler C. Towards a scalable and robust DHT. *Theory of Computing Systems*, 2009, 45(2): 234-260.

[25] Qian Y, Li X, Ihara S, Zeng L, Kaiser J, Süß T, Brinkmann A. A configurable rule based classful token bucket filter network request scheduler for the Lustre file system. In *Proc. the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2017, Article No. 6.

[26] Weil S A, Brandt S A, Miller E L, Maltzahn C. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proc. the 2006 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2016, Article No. 122.

[27] Egwutuoha I P, Levy D, Selic B, Chen S. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 2013, 65(3): 1302-1326.

[28] Artiaga E, Cortes T. Using filesystem virtualization to avoid metadata bottlenecks. In *Proc. the 2010 Design, Automation & Test in Europe Conference & Exhibition*, March 2010, pp.562-567.

[29] Frings W, Wolf F, Petkov V. Scalable massively parallel I/O to task-local files. In *Proc. the 2009 Conference on High Performance Computing Networking, Storage and Analysis*, November 2009, Article No. 22.

[30] Wang T, Mohror K, Moody A, Sato K, Yu W. An ephemeral burst-buffer file system for scientific applications. In *Proc. the 2016 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2016, pp.807-818.

[31] Vef M A, Moti N, Süß T, Tocci T, Nou R, Miranda A, Cortes T, Brinkmann A. GekkoFS—A temporary distributed file system for HPC applications. In *Proc. the 2018 IEEE International Conference on Cluster Computing*, September 2018, pp.319-324.

[32] Zheng Q, Cranor C D, Guo D, Ganger G R, Amvrosiadis G, Gibson G A, Settlemyer B W, Grider G, Guo F. Scaling embedded in-situ indexing with deltaFS. In *Proc. the 2018 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2018, Article No. 3.

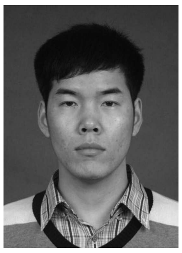
[33] Iskra K, Romein J W, Yoshii K, Beckman P. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proc. the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2008, pp.153-162.

[34] Schmuck F B, Haskin R L. GPFS: A shared-disk file system for large computing clusters. In *Proc. the 2002 USENIX Conference on File and Storage Technologies*, January 2002, pp.231-244.

[35] Grandl R, Kandula S, Rao S, Akella A, Kulkarni J. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. the 12th USENIX Symposium on Operating Systems Design and Implementation*, November 2016, pp.81-97.

[36] Zhou A C, Xiao Y, He B, Ibrahim S, Cheng R. Incorporating probabilistic optimizations for resource provisioning of data processing workflows. In *Proc. the 48th International Conference on Parallel Processing*, August 2019, Article No. 6.

[37] Grandl R, Chowdhury M, Akella A, Ananthanarayanan G. Altruistic scheduling in multi-resource clusters. In *Proc. the 12th USENIX Symposium on Operating Systems Design and Implementation*, November 2016, pp.65-80.



Qi Chen currently is a Ph.D. student of the Department of Computer Science and Technology at Tsinghua University, Beijing. He received his Master's degree in computer architecture from Huazhong University of Science and Technology, Wuhan, in 2013. His research interests include storage architecture for high-performance computing and distributed file systems.

Kang Chen received his Ph.D. degree in computer science and technology from Tsinghua University, Beijing, in 2004. Currently, he is an associate professor of computer science and technology at Tsinghua University, Beijing. His research interests include parallel computing, distributed processing, and cloud computing.

Zuo-Ning Chen received her Master's degree in computer application technology from Zhejiang University, Hangzhou, in 1999. She is an adjunct professor in computer science and technology, Tsinghua University, Beijing, and an academician of the Chinese Academy of Engineering. Her current research interests include big data computing, cloud computing, and high performance computing. She has made important contributions in the field of computer software and high-end computers and received the Special and First Prizes of the National Science and Technology Progress Award of China.

Wei Xue received his Ph.D. degree in electrical engineering from Tsinghua University, Beijing, in 2003. Currently, he is an associate professor of computer science and technology at Tsinghua University, Beijing. His research interests include high-performance computing, and uncertainty quantification.

Xu Ji received his Ph.D. degree in computer science and technology from Tsinghua University, Beijing, in 2019. Currently, he is a senior researcher at Sensetime Inc. His research interests include storage system and distributed system.

Bin Yang currently is a Ph.D. student of High Performance Computing and Big Data Processing Laboratory, the Department of Software Engineering at Shandong University, Jinan. He received his Master's degree in computer science and technology from Shandong University, Jinan, in 2018. His research interests include I/O monitoring, data analysis, and storage.