Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

ONFS: a hierarchical hybrid file system based on memory, SSD, and HDD for high performance computers*

Xin LIU‡[1,2], Yu-tong LU[1], Jie YU[3], Peng-fei WANG[3], Jie-ting WU[2], Ying LU[2]

([1]*School of Computer, National University of Defense Technology, Changsha 410073, China*)

([2]*Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln 68588, USA*)

([3]*National Supercomputing Center, Tianjin 300457, China*)

E-mail: xliu@cse.unl.edu; ytlu@nudt.edu.cn; {yujie, wangpf}@nscc-tj.gov.cn; {jwu, ylu}@cse.unl.edu

Received Sept. 25, 2017; Revision accepted Dec. 25, 2017; Crosschecked Dec. 27, 2017

Abstract: With supercomputers developing towards exascale, the number of compute cores increases dramatically, making more complex and larger-scale applications possible. The input/output (I/O) requirements of large-scale applications, workflow applications, and their checkpointing include substantial bandwidth and an extremely low latency, posing a serious challenge to high performance computing (HPC) storage systems. Current hard disk drive (HDD) based underlying storage systems are becoming more and more incompetent to meet the requirements of next-generation exascale supercomputers. To rise to the challenge, we propose a hierarchical hybrid storage system, on-line and near-line file system (ONFS). It leverages dynamic random access memory (DRAM) and solid state drive (SSD) in compute nodes, and HDD in storage servers to build a three-level storage system in a unified namespace. It supports portable operating system interface (POSIX) semantics, and provides high bandwidth, low latency, and huge storage capacity. In this paper, we present the technical details on distributed metadata management, the strategy of memory borrow and return, data consistency, parallel access control, and mechanisms guiding downward and upward migration in ONFS. We implement an ONFS prototype on the TH-1A supercomputer, and conduct experiments to test its I/O performance and scalability. The results show that the bandwidths of single-thread and multi-thread 'read'/'write' are 6-fold and 5-fold better than HDD-based Lustre, respectively. The I/O bandwidth of data-intensive applications in ONFS can be 6.35 times that in Lustre.

Key words: High performance computing; Hierarchical hybrid storage system; Distributed metadata management; Data migration

https://doi.org/10.1631/FITEE.1700626 **CLC number:** TP303

1 Introduction

Supercomputers deploy parallel storage systems to accommodate the tremendous amount of application data, such as Lustre (Wang *et al.*, 2010), general parallel file system (GPFS) (Schmuck and Haskin, 2002), and parallel virtual file system (PVFS) (Carns *et al.*, 2000). Parallel storage systems have provided high aggregated bandwidth to applications by offering parallel access to their hundreds of storage servers. Hard disk drive (HDD) has been used widely in building parallel storage systems due to its low price and large storage capacity (Seagate, 2017). The capacity of a single HDD-based storage system has reached dozens of PBs (NERSC, 2017b). Although redundant array of independent disks (RAID) has been leveraged to improve the performance of storage systems, HDD's costly overhead of disk-head movements prevents it from meeting the extreme amount

‡ Corresponding author

* Project supported by the National Key Research and Development Program of China (No. 2016YFB0200402)

ORCID: Xin LIU, http://orcid.org/0000-0003-3824-1726

of input/output (I/O) and low I/O latency requirements of future exascale systems (Yildiz *et al.*, 2016).

Nowadays, supercomputers are developing rapidly towards exascale with an increasing number of central processing unit (CPU) cores. For example, TaihuLight, the 1st-ranked supercomputer in the latest TOP 500 List (http://www.top500.org), has about 10 million general purpose CPU cores. The rapidly growing amount of concurrent I/O from compute cores poses a serious challenge to storage systems. Meanwhile, the growing computing power enables scientists and engineers to solve and undertake more and more complex scientific problems and engineering simulations. The data sets processed in scientific research continue to grow dramatically, including research areas of earth (Qiao *et al.*, 2013), gene (Ocaña and Oliveira, 2015), universe (Kuhlen *et al.*, 2012), etc. Taking a typical petroleum seismic exploration application named 'reverse time migration (RTM)' (Dai *et al.*, 2011) as an example, its total input is about 5.08 TB and the total amount of I/O traffic is up to 2433 TB during execution. These data-intensive applications require substantial I/O bandwidth of storage systems. Workflow applications (Bharathi *et al.*, 2008) aggravate such I/O pressure. The latter sub-applications need to read and process the written-out data of the former sub-applications, causing frequent data movements between compute nodes and storage systems (Shibata *et al.*, 2010). In addition, with the increasing scale of applications, they need to write larger checkpoint data more frequently (Sato *et al.*, 2014). This generates an enormous amount of write traffic to storage systems (Rajachandrasekar *et al.*, 2013). Much work (Ali *et al.*, 2009; Dongarra, 2010; Shalf *et al.*, 2010; Bent *et al.*, 2012; Lofstead *et al.*, 2016) has shown that current HDD-based storage systems have been stretched to their limits in handling the tremendous amount of I/O.

To alleviate the I/O pressure on storage systems, many leadership supercomputers insert an I/O forwarding layer to the current high performance computer (HPC) I/O stack, such as BlueGene/Q series (IBM, 2017), Tianhe-2 (Liao *et al.*, 2014), Cori (NERSC, 2017b), and Mira (ALCF, 2017). I/O requests from compute nodes are forwarded to I/O nodes, and I/O nodes access data in storage systems on behalf of compute nodes. Since the ratio of compute nodes to I/O nodes varies from 16:1 to 128:1 (IBM, 2017), the number of clients served by the storage system reduces from tens of thousands (i.e., the number of compute nodes) to hundreds (i.e., the number of I/O nodes). However, this architecture segregates compute nodes from underlying storage systems with an I/O forwarding layer and makes the I/O path longer. As a result, the data movements between compute nodes and storage system must traverse the network and the I/O nodes. The long I/O latency brought by the deepening storage hierarchy impacts the achieved I/O performance.

Flash-based solid state drive (SSD) has become a research focus over the past few years due to its high performance and low power consumption (Vetter and Mittal, 2015). As shown in Table 1, the performance of SSD is orders of magnitude better than HDD. There should be a position for SSD between HDD and dynamic random access memory (DRAM) in the traditional storage hierarchy. A straight forward approach to integrate SSD into a current HPC I/O stack is to build a hybrid storage server with HDD and SSD (Soundararajan *et al.*, 2010; Facebook, 2013; Zhao and Raicu, 2013). However, this approach does not change the storage architecture segregating computing and storage; hence, the I/O bottleneck remains.

Table 1 Performance comparison of storage devices

Device	Access speed		Latency	Capacity
	Read	Write		
HDD	249 MB/s	225 MB/s	2–5 ms	≤10 TB
SSD	3200 MB/s	1575 MB/s	Read: 82 μs; write: 30 μs	≤4 TB
DDR4	63 GB/s	75 GB/s	50 ns	≤128 GB

HDD: hard disk drive; SSD: solid state drive; DDR4: double-data-rate fourth generation

Dongarra (2010) estimated that future exascale supercomputers will require approximately storage system bandwidth of 60 TB/s to meet checkpointing demands. To the best of our knowledge, the currently largest storage system deployed on Cori provides only 700 GB/s peak bandwidth (NERSC, 2017b), which is 100-fold lower than the requirement. Furthermore, approaching this by providing a high-bandwidth external storage system will be very expensive and likely be underused much of the time due to bursty I/O patterns. A survey on Intrepid reveals that during 98% of the time, the storage system is used at less than 33% of peak bandwidth (Carns

et al., 2011). Therefore, the majority of work on improving the performance of HPC I/O focuses on studying storage tiers in the higher storage hierarchy. The storage tiers closer to compute nodes are provisioned for bandwidth, while the underlying storage systems are provisioned for capacity and resiliency (Ovsyannikov *et al.*, 2017).

The US Department of Energy and seven leading US national laboratories initiated a project named 'fast forward storage and I/O (FFSIO)' (Lofstead *et al.*, 2016) to develop an I/O stack suitable for extreme-scale systems. It suggests processing the enormous amount of concurrent I/O in a staging area closer to compute nodes, so that applications can return to computation as soon as possible. The storage architecture in FFSIO is shown in Fig. 1. HDD is left out of the consideration of building fast data staging areas, due to its low performance and reliability. In FFSIO, there are multiple SSD-based data staging areas located in the higher hierarchy of the I/O stack, including the SSDs in I/O nodes and compute nodes. As a step of locating fast SSD devices closer to compute nodes, a burst buffer (Liu *et al.*, 2012) was used to quickly absorb bursty I/O traffic from compute nodes by installing SSD on I/O forwarding nodes. The burst buffer has been proven to be a cost-effective solution and deployed on many HPC facilities (NESRC, 2017a; Schenck *et al.*, 2017). Although there is a budget pressure in equipping every compute node with an SSD (Lofstead *et al.*, 2016), both academia and industry are exploring the expansion of the deployment scale of SSD devices by installing them on part of the compute nodes as the price of SSD decreases. Much recent work has been devoted to managing the node-local SSD (Wang *et al.*, 2016; Yu *et al.*, 2017). Cray DataWarp (Ovsyannikov *et al.*, 2017) uses a temporal file system to manage burst buffer nodes (i.e., compute nodes with SSD) in Cori. Users can use the large temporary storage space to accelerate their applications by manually staging their data in and out.

In supercomputers, as the computing capability of each compute core grows, the average memory used by each core increases accordingly. For example, TaihuLight has a memory of 32 GB per node and a total memory capacity of 1.31 PB. However, not all applications running on supercomputers are memory-intensive; hence, there can be a lot of underused memory in a large production system. We

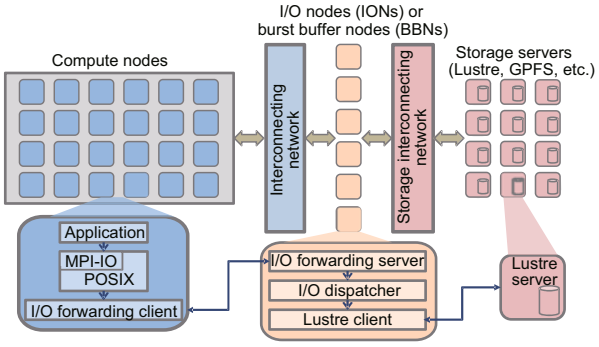

Fig. 1 Storage architecture of future extreme-scale supercomputer

MPI: message passing interface input/output; POSIX: portable operating system interface; GPFS: general parallel file system

conducted a system-wide investigation on TH-1A, and found that about half of the compute nodes have more than 50% free memory. This part of idle memory, which consumes a significant power and has not been effectively used, is a huge waste.

In this paper, to rise to the I/O challenge of exascale supercomputers, we construct a hierarchical file system, on-line and near-line file system (ONFS), to manage both the SSD-based data staging area near the compute nodes and the underlying HDD-based storage system, so that applications can transparently have a high performance and a huge storage capacity. Furthermore, we construct a faster data staging area which is located closer to compute nodes with the borrowed underused memory in the system. Its scale can be dynamically adjusted depending on the I/O workload. We call the memory-based and SSD-based storage tiers the 'on-line storage system'. With the SSD-based tier complementing the limited storage space, the memory-based storage tier seamlessly provides an extremely high performance for applications. The HDD-based storage tier is a near-line storage system that provides mainly tremendous storage capacity. The on-line and near-line storage systems are managed in a unified namespace. We further propose various techniques to enhance the performance of the on-line and near-line file system (ONFS). Our main contributions are as follows:

1. The architecture of a three-level hierarchical hybrid storage system (ONFS)

We leverage memory, SSD, and HDD to constitute a three-level storage system, each of which has different I/O bandwidth and storage capacities. ONFS can provide high bandwidth and a low

latency with a memory-based storage tier, and a huge storage capacity with an HDD-based storage tier. The SSD-based storage tier bridges the performance and capacity gaps between memory- and HDD-based tiers.

2. Distributed metadata storage and management (user group sub-directory (UGSD))

We propose a method to distribute and manage metadata based on UGSD. It preserves the tree structure and locality inherent in metadata directories, and does not generate metadata migration during dynamic workload adjustment. It can effectively support data migration between storage tiers.

3. Memory management and grouping parallel (MM-GP) access control

We propose the MM-GP method to solve the following key issues: (1) memory borrow and return; (2) 'write' operation control under different available storage space; (3) reliability of volatile memory storage; (4) grouping multiple memory-based storage servers together to achieve a larger storage space and higher parallel I/O bandwidth.

4. File coolness measurement (FCM) based on file open-close status and least recently used (LRU) features

We implement data migration in file granularity and introduce file coolness to identify the files for migrating. We set up thresholds of available memory capacity in storage servers to trigger downward migration. The FCM method leverages metrics such as file access status in applications to guide migration.

5. Active upward pre-migration (AUPM)

We propose an AUPM migration policy based on characteristics, in which user programs access and process files. With the set of APIs we developed, users can manually designate which file to migrate. We further leverage the file access logs in previous runs of applications to guide the file migration in the later runs.

2 Background and related work

2.1 High performance computing storage architecture

In the recent development history of supercomputers, three systems stand out as notable milestones. TH-1A (Yang *et al.*, 2011) adopts heterogeneous architecture and achieves an extreme computing performance by leveraging CPU and GPU. Since that time, design focus has been mainly on the HPC storage architecture. IBM BlueGen/Q Sequoia (LLNL, 2012) first introduced an I/O forwarding layer in HPC I/O stack. I/O nodes access data on behalf of compute nodes, alleviating the tremendous amount I/O concurrency faced by storage systems greatly. Cray Cori (NERSC, 2017b) is a recent supercomputer deployed in Lawrence Berkeley National Laboratory (LBNL). It further installs SSD on burst buffer nodes and provides storage space fast to applications dynamically. The enormous amount of I/O traffic is absorbed in the data staging area near compute nodes.

Parallel file systems have been adopted widely by supercomputers because of their features of global data sharing, high performance, linear scalability, high availability, and easy management. Lustre and GPFS are the two most popular parallel file systems in the recent TOP 500 List (http://www.top500.org). In these parallel file systems, metadata flows are separated from data flows so that applications can access data without query metadata in every I/O request. Data is striped across all the storage servers to offer a high aggregate bandwidth. However, HDD-based parallel file systems also have some disadvantages, such as high I/O latency to retrieve data and low performance of processing large numbers of small files. The costly disk-head movements of HDD make it hard for HDD-based storage systems to meet the I/O requirements of exascale systems.

The US Department of Energy proposed a hierarchical storage architecture for future exascale applications in its FFSIO project (Lofstead *et al.*, 2016), as shown in Fig. 1. To reduce the concurrency faced by underlying storage systems, compute nodes are segregated from storage nodes by an I/O forwarding layer. I/O requests from compute nodes are forwarded to I/O nodes, and I/O nodes access data on behalf of compute nodes. Since only I/O nodes can access the underlying storage system, the number of storage system clients is greatly reduced (Iskra *et al.*, 2008). Furthermore, by installing SSDs in I/O nodes as burst buffers, applications can dump data in the staging area closer to compute nodes and return to computation as soon as possible (Liu *et al.*, 2012). As an aggregation point in the I/O path, I/O nodes can reschedule and aggregate small or random

requests into large sequential ones, and further promote the cache efficiency inside the nodes. Most leadership supercomputers adopted the hierarchical architecture, such as Tianhe-2 (Liao *et al.*, 2014), Cori (NERSC, 2017b), and Mira (ALCF, 2017).

Because of the proven benefits of burst buffers and the declining price of SSD, researchers have been exploring the possibility of expanding the scales of SSDs by installing them on compute nodes (Wang *et al.*, 2016). There is still a budget pressure in equipping an SSD on each of the hundreds of thousands of compute nodes. Therefore, installing SSDs on part of the compute nodes is a more feasible approach (Yu *et al.*, 2017). Hence, the majority of work on promoting HPC I/O performance has been focused on studying the fast storage tiers close to compute nodes. However, the use of the node-local fast storage devices has not been well-studied yet, and there is no standard software interface across systems. Hence, how to effectively use the node-local fast storage devices is still an open and hot research topic.

2.2 Hybrid storage server

Much work has been devoted to improving the performance of storage servers by installing SSD on them. A straight forward approach is to replace part or all of the HDDs on the server with SSDs. Many storage vendors have introduced such products. For example, NetApp uses SSDs to construct RAID drivers (NetApp, 2016). EMC also introduced its SSD-based RAID storage system (Dell EMC, 2017).

Another approach is to use SSD as a cache of HDD (Canim *et al.*, 2010; Soundararajan *et al.*, 2010; Cheong *et al.*, 2011). Flashcache (Facebook, 2013) is a block-level SSD cache middleware that effectively improves HDD-based storage servers. It has been deployed on Facebook time line servers and can reduce over 50% write traffic to HDD. HyCache (Zhao and Raicu, 2013) is a user-space caching middleware that uses SSD devices to cache data in the data nodes of HDFS. Flashtier (Saxena *et al.*, 2012) is an SSD cache that provides a unified logical address space to reduce the cost of cache block management within both operating system (OS) and SSD. Hystore (Chen *et al.*, 2011) manages both SSDs and HDDs each as one single block device with minimal changes to OS kernels.

The hybrid storage server approach can boost the performance of storage systems simply by replacing backend storage devices, without introducing too many changes to existing storage architecture. However, there are some limitations in placing SSDs at the lowest end of the storage hierarchy.

First, although the performance of storage systems can be enhanced with SSD, the segregation of computing and storage remains, and can hardly rise to the I/O challenge of exascale systems. Second, as shown in Table 1, today's NVMe SSD delivers up to bandwidth of 3000 MB/s with an I/O latency as low as 30 μs (Intel, 2017). It is becoming a research focus in integrating NVMe SSD into HPC I/O stack because of its declining price and increasing acceptance. However, the network latencies and software overhead in the long I/O path between each pair of compute node and storage node can seriously degrade the performance of NVMe SSD if placed in storage servers. The hybrid device solution cannot effectively deliver the performance benefits of NVMe SSD. Third, the storage servers are far fewer than the compute nodes in a supercomputer. Therefore, the deployment scale of hybrid storage servers is much smaller than that in placing SSDs in compute nodes, resulting in a limited parallel performance.

The above limitations of hybrid storage servers have driven academia and industry to explore the potential of integrating SSDs in higher storage hierarchy of an HPC I/O stack (Lofstead *et al.*, 2016).

2.3 Solid state drive based and memory-based storage tiers

With the growing amount of memory in compute nodes and increasing acceptance of node-local SSD, there has been a rapidly expanding amount of work on integrating fast storage with a current HPC I/O stack. There are two ways to organize the fast storage space: file system and cache.

Much work has extended conventional file systems or proposes novel file systems to organize fast storage near compute nodes. Uta *et al.* (2016) proposed an in-memory runtime file system named memFS, which uses memory in dedicated storage servers to store data. FusionFS (Zhao *et al.*, 2014) is a user-level file system that leverages memory in compute nodes. It distributes file data and metadata to nodes with a hashing function, and migrates jobs to the node where the data resides. BurstFS

(Wang *et al.*, 2016) is a temporal file system that has the same life cycle as a batch-submitted job. It uses a distributed key-value store to manage metadata and supports scalable and efficient aggregation of I/O bandwidth from node-local burst buffers. Cray DataWarp (Ovsyannikov *et al.*, 2017) also uses temporal file systems to manage the burst buffer nodes with local SSD. It strips files across multiple SSDs and supports on-demand apportioning of burst buffer nodes for user jobs.

Caching is an alternative approach for organizing the node-local fast storage. Congiu *et al.* (2016) proposed a set of message passing interface I/O (MPI-IO) hints to cache data in locally attached fast storage. Dong *et al.* (2011) used SSDs in compute nodes as write-back cache to store huge numbers of checkpoint data. These two studies do not relate to general purpose cache systems; thus, they do not need to consider the cache consistency issues. Holland *et al.* (2013) pointed out that the coherency maintenance of a distributed cache system can seriously degrade its performance. NetApp Mercury (Byan *et al.*, 2012) avoids the costly consistency issues by keeping no dirty data in its cache. Many other distributed cache systems solve the consistency issue by establishing a system-level metadata management agency, such as BurstMem (Wang *et al.*, 2014), SFDC (Dong *et al.*, 2015), and WatCache (Yu *et al.*, 2017). These systems make the I/O requests query metadata through a hash-based strategy before accessing data. This ensures that all requests access up-to-date data.

Therefore, using the file system and distributed cache to manage the memory or SSD in compute nodes are just two different design options, since they all need to keep track of the data location by maintaining metadata. In this study, as we intend to manage three different kinds of storage media in HPC systems, we choose to use a file system to implement our ideas for unified management.

In production HPC systems, since not all jobs are memory-intensive, we find that more than 1/4 of the total memory is idle on TH-1A most of the time. Therefore, we intend to make full use of the idle memory and construct a fast storage tier closer to the applications. The premise of our utilization of idle memory in a compute node is that we cannot affect the normal running of the original jobs on that node. To the best of our knowledge, current memory-based storage systems either use dedicated memory servers or have not considered this issue. Therefore, we propose a memory borrow and a return strategy to minimize such impacts to the jobs.

2.4 Hierarchical storage system and data migration

As the number and size of files on a storage system grow rapidly (Agrawal *et al.*, 2007), the hierarchical storage system has become an effective solution to increasing storage efficiency while offering a high I/O performance. For example, Facebook designs a binary large objects (BLOB) storage system which isolates warm BLOBs from hot BLOBs and stores them in a separate storage hardware (Muralidhar *et al.*, 2014). Prabhakar *et al.* (2011) proposed a multi-tiered data staging area in Jaguar Oak Ridge National Laboratory (ORNL), using node-local resources such as DRAM and SSD. It seamlessly uses these devices under different resource contribution constraints and offers an excellent I/O throughput to checkpointing applications. It differs from our work in dedicating a percentage node-local devices to form an in-job staging pool. In contrast, ONFS is a global hierarchical file system provided for all jobs.

In hierarchical storage systems, data migration is the key to offering a high I/O performance of fast storage hardware at the cost of slow storage hardware (Anderson *et al.*, 2001). Most existing work focuses mainly on migration between HDD and SSD (Cheong *et al.*, 2011; Appuswamy *et al.*, 2012), including topics such as migration granularity, data heat identification, static data placement, and dynamic data migration. Many papers have leveraged the block as the unit of data migration and data-hotness identification (Hitachi, 2010; Chen *et al.*, 2011). Relevant parameters include I/O size, access frequency, etc. With block granularity, high accuracy of data-hotness identification can be achieved; however, the cost of computing data-hotness is high. There are also file-based or extent-based data-hotness identification mechanisms, such as HRO (Saito and Oikawa, 2012) and HybridStore (Kim *et al.*, 2011). Large granularity can effectively reduce the number of metadata records and the computing cost. Data migration can be controlled in static or dynamic ways. Static migration control cannot react to real-time changes, while dynamic migration is triggered by specific conditions (Lee *et al.*,

1999; Tan *et al.*, 2013). Since there are many differences among the structure of storage systems and the workloads, there will be a large difference between choosing parameters for processing data-hotness and trigger conditions and choosing control strategies of migration. It is necessary to determine the technical method based on the application environment of the storage system.

Our migration approach differs from the aforementioned related work as follows: We conduct data migration at the granularity of file based on our investigation of access patterns of real applications running on TH-1A. We leverage file attributes such as file lifetime (open-to-close time) and file activity degree (close-to-open time) to assess the value of the file to be migrated. We further develop a set of APIs to allow users manually specifying the files to be discarded and those to be migrated to upper tiers. We find that a class of users run programs under a periodic pattern. Therefore, we leverage the file access logs in previous runs to guide the file migration in the latter runs.

2.5 Distributed metadata management

Since the single metadata server can easily become the bottleneck in a large-scale distributed file system, a lot of literature has studied the distributed metadata management. The following methods are on the storage and management of distributed metadata: The first is static subtree partitioning. The directories of a file system are partitioned based on the directory subtrees, and each metadata server manages one or more sub-directories. NFS (Pawlowski *et al.*, 1994) and Coda (Satyanarayanan *et al.*, 1990) use this strategy. Its problem is that when a large number of requests access the same directory subtree, the corresponding metadata server (MDS) may become the bottleneck.

The second is dynamic subtree partitioning proposed by Weil *et al.* (2006) first. It divides file directories at subtree granularity and allows nested subtree delegates to different MDSs under different subtrees. When a directory becomes a hot spot, the directory is delegated dynamically to a lighter MDS by copying and transferring metadata to achieve dynamic load balancing. It may cause problems of metadata migration and introduces a large maintenance overhead for MDS consistency.

The third is the hash-based metadata service, which uses a hashing algorithm to distribute file directories and metadata to different MDSs. It not only provides better load balancing by leveraging multiple MDSs, but also offers high metadata performance since clients can quickly retrieve metadata by mapping file path to data location. The hash-based metadata service has been adopted by many file systems, such as FusionFS (Zhao *et al.*, 2014), BurstFS (Wang *et al.*, 2016), and zFS (Rodeh and Teperman, 2003). Its problem is that the relationships and locality of directories are corrupted, and directory operations (such as 'ls') are costly. In addition, when the number of nodes changes due to returning borrowed memory, files need to be migrated and the mapping function of hashing needs to be adjusted, resulting in a significant overhead. Hence, the hash-based metadata service is not suitable for our hierarchical storage system which has a frequent file migration among memory-based nodes.

In this study, we incorporate the directory information of real users in TH-1A to construct our scalable distributed metadata service. We use a metadata server cluster to balance the metadata loads. Different metadata servers handle metadata queries of different subsets of users. Since different users have no authority to access the directories of other users, any metadata modification (e.g., renaming file or directory) made by a user happens inside a single metadata server. It significantly reduces the maintenance of metadata consistency in the metadata server cluster.

3 Overview of ONFS design

3.1 Architecture of ONFS

ONFS consists of three storage tiers (memory, SSD, and HDD) and metadata clusters. The first and second storage tiers are composed of memory and SSD from compute nodes, respectively. HDD is configured and managed by a dedicated storage server to build the third storage tier. The metadata clusters are configured specifically. The system architecture of ONFS is shown in Fig. 2, and the main components of ONFS are shown as follows:

Node: compute node in the system, including mainly CPU, memory, and network interface card (NIC). Some of them may also include SSD;

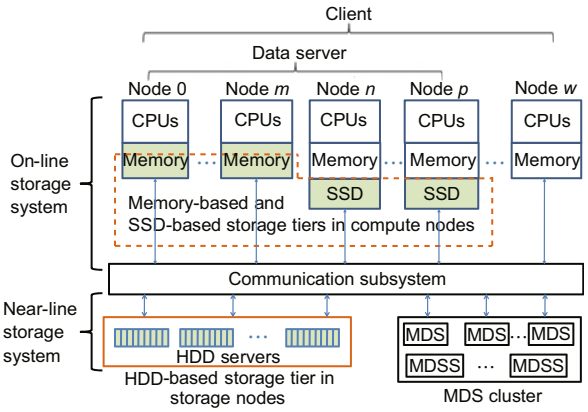

Fig. 2 Architecture of the on-line and near-line file system (ONFS)

SSD: solid state drive; HHD: hard disk drive; CPU: central processing unit; MDS: metadata server; MDSS: metadata storage server

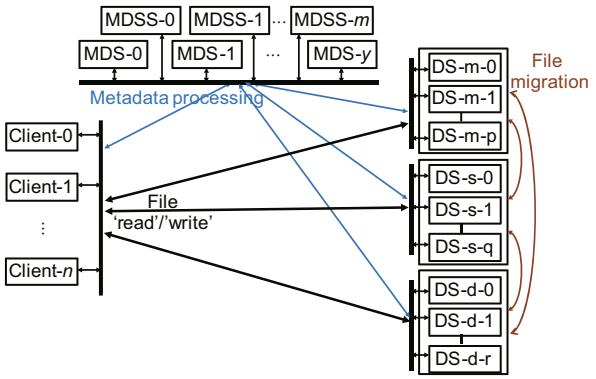

Fig. 3 Logical structure of on-line and near-line file system

HDD server: ordinary server that includes a group of HDD and NIC;

MDSS and MDS: metadata storage server and metadata server.

The main functional units of ONFS are as follows:

MDS cluster: It consists of a set of MDSs and MDSSs. MDS and MDSS are responsible for metadata processing and metadata storage, respectively. All MDSs and MDSSs are dedicated servers.

Data server (DS): There are three types of data server. One is the memory-based data server that distributes in computer nodes, labeled as DS-m. Another is the SSD-based data server in computer nodes, labeled as DS-s. The third is the HDD-based data server, DS-d, which controls dozens of HDDs. DS-m and DS-s are logical data servers.

Client: It is the software running on compute nodes. Users can access ONFS through Client software to read and write data.

3.2 Logical structure and workflow of the on-line and near-line file system

Fig. 3 illustrates the logical structure and the workflow among various units in ONFS. We implemented ONFS in Linux user level with a unified namespace, and ONFS supports portable operating system interface (POSIX) semantics. Therefore, user applications can access ONFS just like accessing local file systems mounted on Linux without modifying programs. DS-m and DS-s tiers provide file accesses with high bandwidth and a low latency. The DS-d tier provides a huge storage capacity.

File requests generated by user programs are forwarded to Client via virtual file system (VFS) and filesystem in userspace (FUSE). To open or create a file, Client first accesses MDS to obtain or create file metadata, which includes file id, data distribution information (DS id), etc. The metadata will be cached by Client for subsequent uses. After that, Client sends the 'read'/'write' requests to the corresponding DS to 'read'/'write' data. A file is stored in one storage tier, and can be kept in two tiers during the migration. Client can access any DS, and files can be migrated between any two tiers.

The research topics in implementing ONFS include distributed metadata storage and management, memory borrowing and return, the reliability of storage, the grouping mechanism of data servers and parallel access control, file coolness measurement, file migration control, etc.

3.3 Data consistency in ONFS

In parallel file systems, multiple clients can access shared data simultaneously through the network. To support the vast majority of HPC applications, ONFS is designed to be POSIX-compliant. According to the UNIX POSIX semantics, any modification of a file made by a process should soon be perceptible by other processes. Lock is the most popular mechanism for implementing concurrent access control.

Some distributed file systems avoid the intricate distribute lock management by getting rid of the client-side cache (Gluster, 2017). However, some HPC applications issue lots of small I/O requests,

such as applications that solve high-dimensional matrix problems (Yu *et al.*, 2017). The client-side cache can significantly accelerate the small I/O. Therefore, we decided to keep the client-side cache, and used a distributed lock management mechanism to maintain cache consistency.

As ONFS is implemented with FUSE, requests from applications first go into VFS via system calls. Then VFS forwards the requests to ONFS Client through kernel FUSE. VFS caches data in the kernel page cache. However, the user-space ONFS cannot control the kernel page cache. As a result, ONFS can neither discard a stale page nor flush a dirty page when a data inconsistency occurs. To address this, we disable the page cache by mounting ONFS with option direct_io. All requests from applications will not be cached in kernel page cache when traversing through VFS. Instead, we design a user-space cache in ONFS Client to cache the application data.

The designed cache has several optimizations on the basis of the kernel page cache. First, the kernel page cache is designed mainly to cache data in local storage. Hence, it has a small cache granularity (4 KB). However, transferring small data blocks over HPC fabrics is very costly since the transferring time accounts for only a small part of the total I/O time and software latency is the main overhead. We increase the cache granularity to improve the caching efficiency. Second, FUSE divides a large I/O request into multiple small requests with sizes no larger than MAX_size, and sends them to ONFS successively. Our cache prefetches data more aggressively to efficiently serve large file I/O.

The distributed lock management mechanism of ONFS is developed based on Lustre DLM. It contains two kinds of lock, a metadata lock handled by MDS and a data lock handled by DS-i. The metadata locks resolve the metadata conflicts occurring in file metadata operations such as 'open', 'close', and 'create'. The data locks resolve data conflicts occurring in concurrent data accesses. DS-i maintains a namespace and locks of data objects stored in it. Any 'read' or 'write' operation in clients must apply for a lock in DS-i in advance. When inconsistency occurs, DS-i will inform the corresponding client to discard stale data or flush dirty data, so that any client can retrieve up-to-date data.

4 User group sub-directory: distributed metadata storage and management

File systems store file data and metadata. There are three important factors indicating that distributed metadata storage and management are essential. First, the total amount of metadata is increasing rapidly. The ratio of metadata size to file size is usually between 0.1% and 1.0% (Miller *et al.*, 2011). When the amount of files increases to dozens of PBs or even hundreds of PBs, the corresponding metadata size will reach TB scale or even dozens of TBs.

Second, metadata operations account for more than 50% of total file operations (Roselli *et al.*, 2000). Therefore, a single MDS may become an obvious bottleneck. Current Lustre uses a single MDS, and the Lustre Union considers a multi-MDS solution to address this issue.

Last, the future generation of supercomputers will have more than millions of compute cores. Each core generates I/O requests independently, may causing tremendous load pressure on MDS.

In this study, we leverage a scalable distributed metadata management approach from our previous work, named 'UGSD' (Liu *et al.*, 2017b). To make the technical content of this study more complete, we briefly illustrate the main ideas in the study.

Fig. 4 describes how Lustre, named 'Vol6', is mounted under the Linux root directory '/'. ONFS is mounted to the local file system in the same manner.

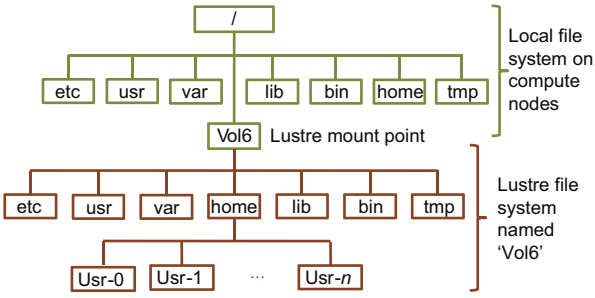

Fig. 4 Basic structure of Lustre and Linux directories

4.1 Metadata division unit

The metadata division unit is the basic to achieve metadata storage and management. We use

Lustre as an example to illustrate user directories. Users in Lustre 'Vol6' create their own directories under Lustre's 'home' directory, which has the highest level among all sub-directories. Each sub-directory, such as 'Usr-i', represents a user group. We refer to the sub-directory as a UGSD. Table 2 represents the statistical data of UGSDs in Lustre 'Vol6' that is deployed on TH-1A.

Table 2 Statistical data of user group sub-directories (UGSDs) in TH-1A Lustre 'Vol6'

Parameter	Value
Total number of UGSDs	1183
Total number of secondary sub-directories	13 608
Number of directory layers for the deepest UGSD	27
Total number of directories in UGSD A	418 246
Total number of files in UGSD A	2 217 243

It shows that there are 1183 UGSDs in total. The maximal directory depth under one UGSD is 27. When an HPC system is running steady, the number of UGSDs does not change much. It is reasonable that there are tens of thousands of UGSDs in the system. We use UGSD as the basic unit for directory partitioning.

4.2 Composition of the metadata system

The composition of the metadata system is shown in Fig. 5. MDS and MDSS are responsible for metadata processing and storage, respectively. Peak-shaving metadata server (MDSps) is responsible for MDS load balancing and MDS scale expansion. DS-mi represents the ith DS-m, and so on.

4.3 Mapping relationship between the user group sub-directory and metadata server

Denote n the value of UGSD. We planned to use a specific algorithm, such as MD5, to transfer the name of UGSD (a serial of characters) to an integer N. Since the total number of UGSDs is not large, the values of N's are not evenly distributed. Therefore, a sequential number n is added after the name of sub-directory, such as Ocean-n, where n is used as the input of the MOD function as it is evenly distributed.

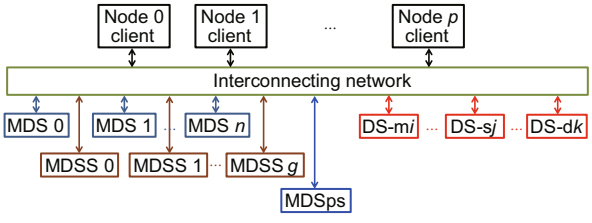

Fig. 5 Composition of the UGSD metadata system
DS: data server; MDS: metadata server; MDSS: metadata storage server; UGSD: user group sub-directory; MDSps: peak-shaving metadata server

To balance the loads of metadata storage, one MDSS provides metadata storage for several MDSs. Since the number of MDSs and MDSSs are not large (usually dozens), we use a look-up table to map MDS to MDSS. The mod function and the look-up table constitute a two-level mapping, shown in Fig. 6. In Table 3, we show a mapping example with eight MDSs and four MDSSs.

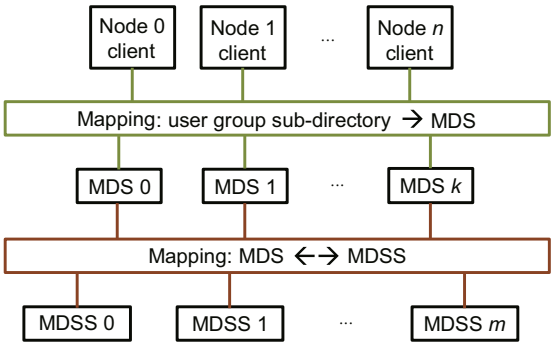

Fig. 6 Two-level mapping relationship
MDS: metadata server; MDSS: metadata storage server

Table 3 Mapping relationship between metadata server (MDS) and metadata storage server (MDSS)

MDS No.	MDSS No.	MDS No.	MDSS No.
0	0	4	0
1	1	5	1
2	2	6	2
3	3	7	3

4.4 Dynamic metadata modification and updating

One of MDSSs is used as duty MDSS that collects metadata modifications from each MDS periodically. We set up some parameters in each MDS to help achieve metadata synchronization and consistency between MDS and MDSS. There are two pointers (P-metaP and P-metaB) pointing to two buffers (buffer0 and buffer1), respectively,

and one synchronization time point Tsyn. Modified metadata is stored in two buffers during the synchronization period to form a latest, complete metadata. When MDSS informs MDS to send back all modified metadata that is generated from the last synchronization time Tsyn to current time T, MDS first sets Tsyn to current time T and clears the buffer pointed by P-metaB. Then MDS exchanges the values of two pointers and sends the data in the buffer pointed by P-metaB to MDSS. New metadata modification records will be saved to the buffer pointed by P-metaP. Hence, we can achieve the synchronization and consistency of metadata between MDS and MDSS. Fig. 7 shows the relationship between pointers and buffers, in which tail0 represents the pointer for data write and tail1 represents the pointer for the last data of read.

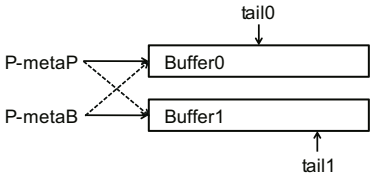

Fig. 7 Relationship between pointers and buffers

After the duty MDSS has finished collecting the latest metadata modifications from all MDSs, it updates the system-wide metadata, and informs other MDSSs and MDSps to obtain modifications of all MDSs and to update their local full metadata copies. In this way, the metadata on all MDSSs and MDSps are consistent. Dynamically updated and complete metadata on MDSps is the basic for dynamic MDS load balancing and MDS scale expansion.

4.5 Load balancing

There are two cases where the loads on MDS may increase. One occurs locally, such as on one MDS. The other happens globally which involves all MDSs. We use the following methods to balance the loads:

When a hot spot of workload appears on MDS-h, metadata requests sending to MDS-h will be redirected from MDS-h to MDSps. After the requests have been fulfilled by MDSps, the result is transferred to the requesting Client. Before starting to transfer the requests, MDSps synchronizes with MDS-h to obtain the modified metadata generated after previous synchronization time point Tsyn.

When the number of the whole workloads on MDS cluster increases, a portion of UGSDs can be taken out of some or even all MDSs and served by MDSps. We express the UGSD suffix number n in binary representation, and divide it into lower k bits and higher $(n-k)$ bits. The partition is shown in Fig. 8.

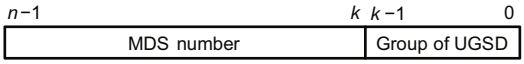

Fig. 8 Segmented diagram of the suffix number n in user group sub-directory (UGSD)

The Client calculates MDSs by using higher $(n-k)$ bits of the suffix number. When adding a new MDS is required, UGSDs mapped to original MDS are divided into 2^k groups according to the lower k bits. Group 0 of UGSD or multiple groups can be transferred to MDSps. Duty MDSS notifies all Clients of the number of groups to be taken out and the command of load balancing. Before starting the load balancing, MDSps synchronizes with each MDS to update metadata.

With the above processes, the UGSD method neither generates metadata migration nor suspends metadata services. It has a good performance in MDS load balancing and shows a good scalability in MDS scale expansion. We have discussed the details in our previous paper Liu *et al.* (2017b).

We then discuss the data migration among storage tiers. It involves maintaining data consistency. We have discussed how to maintain consistency of metadata between MDSS and MDS. Maintaining data consistency includes two cases. Under a non-migrating status, the file is stored in one DS, and its sharing by multiple processes gives a consistency problem. ONFS manages metadata uniformly through the metadata cluster, and supports POSIX. Therefore, we adopt the lock mechanism of Linux-based file systems to maintain the data consistency for shared 'read' and 'write'. For files under migrating status, corresponding methods should be employed to maintain data consistency. We will discuss the main technical methods in Section 7.3.

5 Memory management and grouping parallel access control (MM-GP)

The memory resource used by DS-m belongs to the compute node. It is owned by the user who has the right to access it. ONFS cannot occupy memory by force. It can leverage idle memory only by borrowing it, and has to return the borrowed memory when user programs require. The maximal memory capacity that DS-m can use is just dozens of GBs. Therefore, 'write' operations need to be controlled under different available storage capacities in case that the node runs out of memory and causes the writes to be terminated abnormally. To enlarge the available storage capacity and to improve 'read'/'write' bandwidth, it is necessary to make N DS-m's into one group to provide parallel data access. Since memory is a volatile storage device, it is required to solve the reliability problem of storage. We discuss the above technical issues in Sections 5.1–5.4.

5.1 Memory borrowing and return

5.1.1 Borrowing memory resources

We first discuss how to borrow memory, and how much memory can be borrowed. It is a very important issue that many studies have overlooked.

We have analyzed the memory use of compute nodes (referred to as node) in TH-1A (Liu *et al.*, 2017a) and found that 85% of the CPU-intensive programs use less than 50% of the nodes' memory. The number of nodes that run CPU-intensive program accounts for more than 50% of the total number of nodes. Therefore, we divide the whole nodes into two partitions based on the programs' memory requirements. One is full-memory-partition (FMP), in which programs can use all memory of a node. The other is small-memory-partition (SMP), in which programs can use only some part of the memory resources of a node, such as 1/2. The entire memory-based storage tier can borrow about 1/4 of the total memory resources in the supercomputer.

In SMP, we allow users to use 1/2 of total memory capacity in nodes in an explicit and forced manner. Users are aware of this restriction and they agree to accept. Therefore, user programs running in SMP usually do not apply more than 1/2 of the memory capacity. If the memory requirement of a user program exceeds 1/2 of the restriction, the program will be terminated as out of memory.

5.1.2 Returning memory resources

Once the number of nodes in FMP becomes insufficient to fulfill programs' requirements, ONFS begins to move one or more compute nodes from SMP to FMP. We first select free compute nodes in SMP to move. If there is no free node in SMP, we must wait for users to release nodes. We preferentially sort them based on used memory capacity, and choose nodes that have used the least memory capacity as 'to be returned'. The purpose is to reduce the amount of data migrated. Files stored on the 'to be returned' node will be migrated to other nodes, or to DS-s. As we employ dual-replicas to maintain reliability, the memory return follows the features of dual-replicas, and we move a pair of nodes (primary and secondary nodes) together at one time.

During data migration, the primary node with the original file continuously provides data accesses, and the secondary node migrates the replicas to the new primary and secondary nodes. We first set a synchronous time point T. The 'write' requests to the 'to be returned' node after time T will be stored in a log file in the primary node without replicating to the secondary node. After all data in the secondary node have been migrated to a new pair of nodes, following data accesses to the primary node will be blocked. Then the data in new nodes will be updated by the log file in the old primary node, and the metadata of the mapping between files and DSs will be modified. At this time, blocked Clients will be informed to access MDS again for the information of new DS-m's. The original DS will be closed to all Clients, and any access to the original DS will be timed out. Then they can re-access MDS to obtain the new metadata of the file. Hence, the data migrations in the primary node and the secondary node are completed. After that, we modify the restriction of memory use of 'to-be-returned' node and move the nodes to FMP. If none of the compute nodes have enough memory to store migrated data, the data in 'to-be-returned' node will be moved to DS-s with similar procedures.

Besides static memory borrowing, we can borrow idle memory from 1/2 of the remaining memory reserved for user programs in SMP nodes with dynamic memory borrowing and return approaches.

We have discussed the technical problems in detail in Liu *et al.* (2017a).

5.2 Write management based on available memory space

In current HPC applications, the average size of user files is usually several MBs, and the size of large files can be several GBs. The available storage space of a DS-m is about tens of GBs, and it can be exhausted easily by sharing to all users. When the available storage space becomes insufficient for a 'write' request, the write will be terminated abnormally. It seriously affects the availability of ONFS. Hence, there are two problems which need to be solved. One is how to control the 'write' operation so that it can be processed correctly when a DS-m's available storage space is insufficient. The other is how to expand the available storage space of a DS-m effectively to support large file storage.

The storage space is allocated as follows. After the Client receives the 'write' request, it determines DS by the metadata of the requested file, and accesses DS directly. If the 'write' request requires new storage space, then DS allocates a storage allocation unit (SAU). We choose the size of SAU as 4 MB, and discuss the specific reasons for this in Section 8.6. On the available storage space, there are two related questions. First, the Client does not know whether there is enough storage space in DS-m. Second, DS-m does not know the amount of data to be written for the 'write' operation. If there is not enough storage space in DS-m, in general, the 'write' request will be terminated abnormally. This may seriously affect the availability of ONFS. In this case, we set a threshold Cs-w. When the available storage spaces reduce to Cs-w, if the 'write' request requires SAU, DS first blocks the 'write' request, and resumes the execution until DS-m releases the storage space that can meet the requirements. The relationships between the available storage space and the control of the 'write' operation is shown in Table 4.

To maximize memory utilization, Cs-w is chosen to be near zero. To prevent the available storage space from decreasing to or under Cs-w, we leverage the dynamic migration method to move selected files from DS-m to DS-s to release the storage space. We set two thresholds of available storage space to control the migration for closed and opened files to DS-d and DS-s, respectively (referred to as in Section 6.2).

Table 4 Thresholds for 'write' operations

Type of DS	Available storage space	Operation
DS-m	$\leq C$s-w	Suspending writing and waiting for available storage space
DS-s	$\leq C$s-w	Suspending writing and waiting for available storage space

DS: data server

5.3 Implementations of grouping and parallel access control

Since the maximal available memory space in a DS-m is only tens of GBs, it is difficult to effectively support large file storage. We propose a method that groups multiple DS-m's together to provide parallel data access. We set G-count and G-size parameters. G-count is the number of DS-m's in a Group. Files are divided into blocks with a size of G-size and distributed to DS-m's within a Group in a circular manner.

As the 'read'/'write' bandwidth under a single process reaches 1800 MB/s, we gather four DS-m's within one Group. Hence, the aggregated 'read'/'write' bandwidth of four DS-m's can be 7200 MB/s. Though there exists other overhead, its bandwidth may match Client's external communication bandwidth (the unidirectional bandwidth of NIC is about 5–6 GB/s). In addition, as we choose 4 MB as SAU_size, we set G-count as four and G-size as 4 MB. The values can be adjusted as needed.

As we cannot require user programs to write in strict accordance with the appended write manner, there may be empty segments in a file's logical address space. In each DS-m, we use pointers to connect discontinuous segments in the address space. Hence, when a 'write' operation is completed, the written data in each DS-m in a Group may not be similar. Hence, we manage the storage space of four DS-m's in one Group in an independent manner. DS-m takes the responsibility for allocating storage space for a 'write' request, and all management methods are the same with a non-group pattern.

From the view of the composition of the Group, it is similar to the Lustre striping mechanism. The main reason for our discussion here is that the grouping method faces special problems during the migration, such as how to determine all DS-m's within one Group by DS-m that triggers the migration, and how to control the migration. Grouping and

parallel storage can be used to solve the thorny problem that the available storage space is insufficient due to the limited memory in one DS-m. They also further improve the bandwidth of data access.

5.4 Reliability of DS-m storage

DRAM-based memory is a volatile storage device. In practical use, any breakdown of memory or a compute node will cause loss of the data that is stored in memory in nodes, and this is unacceptable.

There are many ways to improve the reliability of data storage, such as multi-replicas, RAID, and erasure code. DS-m has two obvious features: high 'read'/'write' bandwidth and low available storage space. The memory space occupied by RAID and erasure code is small. However, it takes a long time to calculate the parity and erasure code, which decrease the I/O bandwidth of DS-m dramatically. Though multi-replicas require additional storage space with the same size, a replica can be generated within a short time, and this has little impact on the speed of data access. As a result, we choose dual-replicas to maintain a high reliability.

We design a dual-replicas policy with a primary node and a secondary node, which are distributed to different cabinets. From the experience of running TH-1A, the probability that two cabinets fail simultaneously is very low. The number of compute nodes can be divided into two parts. The lower n bits of the number express the number of compute nodes in the cabinet. In TH-1A, n is 6, indicating that there are 64 compute nodes in one cabinet. The remaining higher bits represent the number of cabinets. To simplify the design of the Client, only the primary node communicates the control information with the Client. The control procedure of a 'write' request is discussed as an example. The Client sends the 'write' request to the primary DS-m, and the primary DS-m forwards the messages to the secondary DS-m. Then the primary and the secondary DS-m's use RDMA_read to obtain the data simultaneously. The primary DS-m waits for the completion of the secondary DS-m. Once the primary DS-m receives the completion report from the secondary DS-m, it replies to the Client. The Client then completes the 'write' request. Fig. 9 illustrates the write process with dual-replicas.

MDS uses a heartbeat mechanism to check the status of DS-m. Client also introduces a timeout

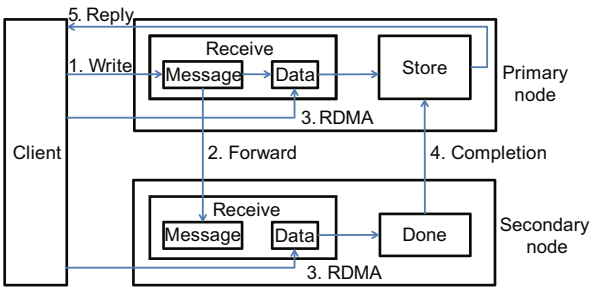

Fig. 9 'Write' operation with dual-replicas

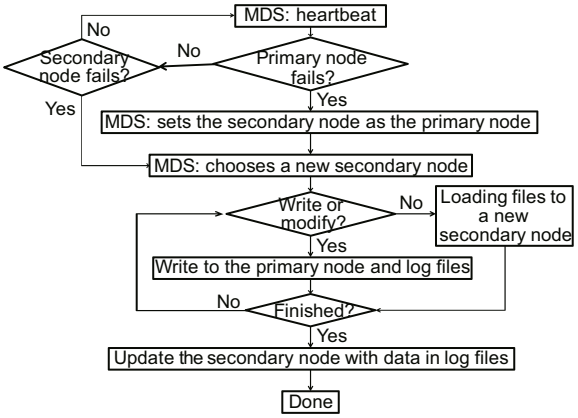

Fig. 10 Recovery procedure when one node crashes (MDS: metadata server)

reporting mechanism. There are three kinds of breakdowns: either the primary or the secondary node crashes, or both nodes crash. The situation that both nodes crash at the same time is a disaster which cannot be recovered from. Fig. 10 shows the recovery procedure when one node crashes.

If the primary node crashes, MDS changes the secondary node to the primary node and modifies the metadata. Then MDS assigns a new secondary node, and backs data up in the new primary node to the new secondary node. During data replication, all newly written data are also written into the new primary node and stored in log files. Once the data replication completes, the primary node uses the data in log files to update data in the secondary node. After that, the dual-replica mechanism continues.

If the secondary node crashes, the 'read' and 'write' operations are processed ordinarily. After choosing a new secondary node, the primary node begins to load data to the new secondary node. The following operations are the same as from step 3 in processing primary node crashes.

If the primary node crashes while processing a 'write' request, the current 'write' request will be terminated abnormally. Then it follows the processing steps of primary node crashes.

Ensuring the reliability of DS-m is a complex issue. Specific and in-depth research work is required.

6 File coolness measurement (FCM) for downward migration

In existing studies, the characteristics of data access, such as access frequency, latency, and file size, have been employed to select files or data to be migrated. The granularity of data migration, such as block, is often fine-grained. This will lead to a great migration control overhead. When the scale of the storage system is large, the aggregated cost is too large to bear. We use the file as the migration granularity based on the characteristics of file processing.

To prevent the 'write' operation from failing due to capacity overflow on DS-m, some files need to be moved out. This is triggered by available capacity threshold. The migration direction and migration conditions are different from the usual methods. We refer to the migration condition as the coolness, not the usual hotness. We determined the factors related to the coolness of the file and proposed a method to measure the coolness.

6.1 Migration granularity

The migration granularity is related mainly to the purpose of migration and the way that users read and write data. We take RTM as an example to analyze how data-intensive applications read and write data. In one instance, RTM processes about 200 000 input files, and the size of each file is about 207 MB. It applies K nodes to process files in parallel (e.g., K is 1024). For the input files, each file is read continuously from the beginning to the end. Other typical examples, such as genetic and climate applications, share the same characteristics. This implies that opened files are usually read in sequence.

Based on the analysis above, taking the file as the unit of migration is reasonable. In ONFS, files are created in DS-m or DS-s to achieve high bandwidth for data accesses. For ease of discussion, we refer to data migration from DS-m to DS-s, from DS-s to DS-d, and from DS-m to DS-d as downward migration. Migration in the opposite direction is considered as upward migration.

Upward migration is triggered by open commands or instructions from user programs. The main purpose of downward migration is to release storage space. We determine the following migration strategies: (1) When migrating out opened files from DS-m, we first choose DS-s as the destination; (2) Closed files will be migrated to DS-d; (3) The files in DS-s will be migrated to DS-d.

6.2 Controlling strategies of downward migration

In upward migration, files to be migrated are determined by user programs. Therefore, it is not necessary to analyze the hotness of the file. ONFS migrates files that are least used recently from an upper storage tier to a lower tier. The condition to choose files is called 'file coolness'. It varies with different available storage space. As shown in Table 5, we set up two thresholds of available memory space, Cn-m and Cf-m, to control the migration. Cn-m is larger than Cf-m. We explain the meaning and role of Cn-m and Cf-m, and set C as the available storage space in DS-m.

Table 5 The control of file migration

Type of DS	Available storage space (C)	Downward migration	Destination
DS-m	Cf-m $< C \leq C$n-m	Normal out	DS-s or DS-d
	$C \leq C$f-m	Forced out	DS-s or DS-d
DS-s	Cf-m $< C \leq C$n-m	Normal out	DS-d
	$C \leq C$f-m	Forced out	DS-d

DS: data server

When Cf-m $< C \leq C$n-m, closed files that satisfy certain conditions will be moved out. We refer to this condition as 'normal out'. When $C \leq C$f-m, opened files that satisfy certain conditions will be moved out, referred to as 'forced out'.

In the calculation of file coolness, we consider static features of files and dynamic characteristics of file operations. Static features include file size, file type, etc. Dynamic characteristics include access frequency (e.g., LRU), file status (open/close), access pattern (random/sequential), operation type ('read'/'write'), etc. Since the factors affecting file coolness in 'normal out' and 'forced out' are different, we discuss these two cases in Sections 6.2.1 and 6.2.2, respectively.

6.2.1 Normal out

In a 'normal out' scenario, only closed files can be selected to be migrated. Files in open status will not be migrated. The other condition is the time for which a file has been closed, referred to as Tclosed. Files with a greater Tclosed are less active. Other factors are the time periods between open and close operations, Tco and Toc. Tco stands for the time between the close and subsequent open operations. Files with greater Tco are used less frequently. Toc stands for the time between open and subsequent close operations. Files with shorter Toc are worth less to be stored.

File size is a factor which is difficult to choose. From the perspective of releasing memory space, migrating files with a larger size can release more space. However, if the file will be reused in the near future, keeping it can bring more performance benefits. Given this contradiction, we do not take file size into consideration. In addition, since only closed files are selected, parameters such as operation type, access pattern, and access size are not considered.

Based on the analysis above, we choose the following expression to calculate file coolness for 'normal out':

$$\text{Close} \cdot T\text{closed} \cdot (T\text{co}/T\text{oc}),$$

where Close is either 0 or 1 to present the file status.

6.2.2 Forced out

To prevent using up the memory space of DS-m's and thereby resulting in a failure of file write, DS-m migrates some files to a lower storage tier to release a certain amount of memory space by force. In 'forced out' status, some opened files need to be migrated to a lower storage tier. We introduce LRU-open queue to record the time order of file accesses after files are opened. Other factors are Tco and Toc. According to the above analysis, the file size is still not considered. We choose the following expression to calculate file coolness for 'forced out':

$$\text{Open} \cdot \text{LRU-open} \cdot (T\text{co}/T\text{oc}),$$

where Open is either 0 or 1 to present the file status. As a queue, we select files from the head of LRU-open.

DS-s follows the same migration policies for 'normal out' and 'forced out'.

6.3 Determination of file coolness

In this subsection, we discuss how to obtain the aforementioned parameters: Tclosed, LRU-open, Tco, and Toc.

To determine Tclosed and LRU-open and select files to be migrated to a lower storage tier, we design two tables for each DS-m and DS-s, the ObjectID table and the LRU table. Fig. 11 shows the composition of tables.

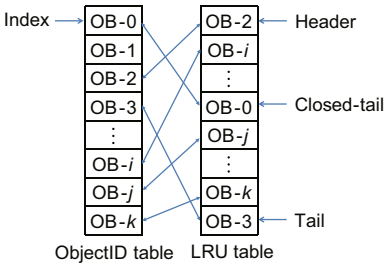

Fig. 11 Compositions of ObjectID table and least recently used table (LRU-table)

LRU table is a double-linked-list which records the status of files (closed/open) and the time sequence of file status. There are three pointers: Header, Closed-tail, and Tail. Files recorded between Header pointer and Closed-tail pointer are closed. Opened files are stored from the next entry of Closed-tail pointer to Tail pointer. For an opened file, its active degree is determined by file reading and writing. Hence, we need to modify the location of the file in the LRU table in time when reading and writing the file. We adjust the position of the file in the LRU table to Tail when the address of a 'read'/'write' request is within the first 1 MB range of an SAU (4 MB size), and we record the address for this adjustment. Files are arranged in descending order of modification time.

ObjectID table maps the ObjectID to the entries of the LRU table. When MDS allocates DS to a file, it assigns an ObjectID to the file on DS. The Client uses this ObjectID to communicate with DS and completes data reading and writing. For ease of management, we generate a finite sequence number OB-i starting from zero on each DS. DS assigns an OB-i to each newly arrived ObjectID, and retrieves OB-i for reallocation when the ObjectID is deleted. OB-i is used as an index in ObjectID table. There is a pair of pointers mapping each OB-i in ObjectID table to the corresponding OB-i in the LRU table.

Tco and Toc relate to the time point when a file is opened or closed, respectively. Since a file is only temporarily stored in DS-m or DS-s, once it is removed from DS-m or DS-s, the historical records of file accesses are cleared. MDS is responsible for calculating Tco and Toc. We calculate the sum of Tco-i and Toc-i. Since current open and close statuses of files are recorded, it is necessary to record only the time of the last open or close. MDS calculates the average of Tco and Toc each time when receiving a file-open or file-close request from the Client. MDS sends the average values of Tco and Toc to the Client, and the Client then forwards the values to DS-m or DS-s.

6.4 Determining the values of thresholds *C*f-m and *C*n-m

The relationships among thresholds are shown in Fig. 12, where Max shows the maximal available memory space in a DS-m, and Cs-w, Cn-m, and Cf-m represent the remaining available memory space.

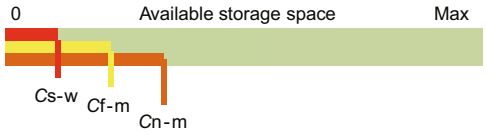

Fig. 12 Relationships among thresholds (References to color refer to the online version of this figure)

According to the above analysis, Cs-w notifies the control module of 'write' operations that there is no available SAU, and asks for suspension of 'write' requests that require new SAUs. Therefore, we can set Cs-w to 0. Cn-m and Cf-m determine when to migrate closed and opened files, respectively. If the threshold is too large, files will be migrated out prematurely and the service quality will be reduced when the migrated file is accessed again. We introduce the quality-of-service (QoS) method to meet the requirement of service quality (Lu *et al.*, 2002). There are two main requirements for service quality: (1) the latency of 'write' request should be short, and (2) the number of migrated files should be small. With thresholds Cs-w, Cf-m, and Cn-m, the available storage space is divided into three regions. Table 6 presents the service quality and corresponding control policy for each region.

Table 6 Determining the quality of services and control policies with available storage space

Region	Quality of service	Control policy
MAX–Cn-m	Speed of writing in	No adjustment
Cn-m–Cf-m	Balancing the speed of writing in and migrating out	Only migrating out closed files, $C > C$f-m
Cf-m–Cs-w	Balancing the speed of writing in and migrating out	Controlling the speed of migrating out and writing in, $C > C$f-m

To meet the control requirements in the above table, the speed of migrating files needs to be managed, and the speed of writing files that apply a new SAU also needs to be controlled.

Taking migrating closed files from DS-m to DS-d as an example, we discuss how to determine Cn-m and manage writing in and migrating out. We leverage 24 independent HDDs to constitute a DS-d. As the 'read'/'write' bandwidth of a single HDD is low, if the number of files to be migrated is large, the control and management of file migration will be more complicated, hence more representative. The input parameters are: (1) 'write' requests from one or multiple Clients that require new SAUs (referred to as "'write' request" in following discussions), and (2) a file list for closed files, which are sorted in descending order of LRU. We choose the following parameters: average file size (S-avg-f), 'write' bandwidth of a single DS-d (B-1w-DSd), and 'write' bandwidth of DS-m (B-w-DSm).

We first show how to determine $\Delta C\text{n} = C\text{n-m} - C\text{f-m}$. The principle is to prevent the speed of 'write' request that applies for SAUs from being affected by available storage space. The number of files to be migrated (N) can be calculated as follows:

$$N = (B\text{-w-DSm})/(B\text{-1w-DSd}). \tag{1}$$

The total size of migrating N files simultaneously is $N \cdot S$-avg-f, which can be set as the initial value of ΔCn. In the same manner, we can determine $\Delta C\text{f} (= C\text{f-m} - C\text{s-w})$. With values of ΔCn, ΔCf, and Cs-w, we can construct a producer-consumer model, as shown in Fig. 13.

When determining the values of thresholds, we assume that there is no 'read' request from user programs, and all 'write' requests are applying for SAU. Obviously, it is an extreme case. Therefore, the

Fig. 13 Producer-consumer model

values of thresholds need to be adjusted dynamically according to real situations.

6.5 Implementations of downward migration

It is not difficult to migrate files that are stored in a single DS-m or DS-s (non-grouped) to a lower storage tier. However, migrating files within a Group is complex since each DS manages files independently, and the downward migration is triggered by one DS which does not know the information of other DSs within the Group. We take the migration from a group of DS-m's to a group of DS-d's as an example to describe the implementations of downward migration.

When a DS-m submits a migration request to MDSS, it provides the information of the ObjectID, such as file name, UGSD, and required storage space. MDSS finds the corresponding MDS based on the mapping function from UGSD to MDS, and sends the migration request with the information to MDS.

The MDS first checks all DS-m's associated with ObjectID and the total memory space it occupied in these Ds-m(s). Then MDS allocates destination DS, such as DS-s or DS-d, based on an available storage space, and constitutes a migration control table. On the source DS, there are the following parameters: number of DSs per Group (G-count), distribution granularity (G-size), and data size in each DS (DS-size-i). The total size of the file (size-file) can be calculated by

$$\text{size-file} = \sum_{i=0}^{G\text{-count}-1} \text{DS-size-}i. \tag{2}$$

We use the same parameters (G-size and G-count) in three storage tiers to construct a Group. In the following discussion, we set G-count as four, and G-size also as four. The migration control table contains mainly four sub-tables. Each sub-table describes the migration control relationships between corresponding source DSi ($i \in \{0, 1, 2, 3\}$) and the destination DS, especially the distribution of data storage in source DSi. The destination DS stores data according to the data distribution requirements provided by source DSi.

Another question is: what should we use to control the migration? We choose the destination DS to control the migration. As the source DSi in the upper tier provides services for data access, it is busy. If any of Client, MDS, or MDSS is chosen to control the migration, data has to forward, introducing much more overhead. MDS sends the migration control table to all relevant DSs. In this example, each DS-d(i) completes its own migration and reports the completion of migration to MDS. After receiving completions from all DS-d's, MDS then modifies the metadata of the file, and instructs DS-m to release memory.

During downward migration, if an opened file is read or written, or a closed file is opened, the migration will be terminated immediately. The cost is occupying some resources during migration, and there is no impact on the performance of file accesses.

7 Active upward pre-migration (AUPM)

Upward migration is to migrate files that are being accessed or will be accessed to an upper storage tier for a higher file access speed. It is performance-critical and time-sensitive. We can migrate files that are being accessed or to be accessed by automatic control or active pre-migration from a lower to an upper storage tier. Researchers have proposed many automatic upward migration methods. When applying these methods in ONFS, proper adjustment is required in line with the workload of ONFS.

7.1 Feature analysis of reading input files

We first analyze the characteristics of user programs in file reading and writing, and then we leverage these features to select the files to be migrated. We classify files into three categories: input file, intermediate result, and final result. Most intermediate and final results are new files created by the program, and they are stored in DS-m or DS-s to avoid upward migration. Input files already exist in the storage system before programs run. They are stored either in the lowest DS-d tier or in users' storage devices, and they need to be migrated to upper storage tiers when programs are running. By analyzing various typical data-intensive applications, we divide the input files into two categories and summarize their corresponding patterns of processing different types of input files.

1. Large numbers of input files: The total number of input files (N) is large, and the size of each file is about several hundred MBs. This kind of application usually applies for K compute nodes, where K is much smaller than N. Taking the Gathering program in Section 8.8.1 as an example, N is 210 000 and K is 512. Each node reads one or multiple entire files independently, and processes them in a complete processing step. After one node finishes processing the current file, it reads and processes the next file.

2. One single input file of large size: In this situation, the input file is dozens of GBs. It is sequentially divided into many sections of smaller size, and these sections are read one by one sequentially. Each time, a section of the file is read and processed. For applications such as gene sequencing, the input file is about 90 GB, and the size of a section is about 2 GB.

In addition, there are other important I/O features, such as the starting point of file reading. In a single run of the user program, each input file is usually read only once. Therefore, if an input file is being read, it is worthless to migrate the file to an upper storage tier as it will not be accessed again. Based on the above analysis, we divide the upward migration into two types. One is active upward pre-migration (AUPM) and the other is passive upward pre-migration.

7.2 Implementations of active upward pre-migration

We propose the following three methods for active upward pre-migration:

1. Directly upload files to an upper storage tier

In applications such as petroleum exploration, weather forecasting, ocean exploration, the amount of data to be processed is large. Taking RTM as an example again, the raw data to be processed is about several TBs. When users prepare to run the program, ONFS uploads the input data from users' own storage devices to either DS-m or DS-s directly.

2. 'On-demand' pre-migrate based on applications' instructions

User programs send a list of files to be processed (including the order in which files are read) to ONFS by particular APIs. When the program runs, each compute node reads one entire file and processes it. The next file is read after the process is completed. With this workflow, we determine the following processing strategies: (1) We determine the first group of files that will be read by all compute nodes based on the file list and the number of nodes. These files will not be migrated as they will be read immediately. (2) We determine the second group (or the third group) of files to be read, and migrate this group of files to an upper storage tier. In addition, we migrate the following groups of files to the upper storage tier sequentially. In this way, following data accesses can be served by the upper storage tier.

3. Start upward pre-migration based on historical records

By analyzing historical records, we find that there is a class of users who run programs in a periodic pattern. For example, weather forecasting program runs at a fixed time every day. The users process the same type of meteorological observational data in the same directory. To save memory space in DS-m, input files are originally stored in DS-d. Before the program runs, we migrate input files to the upper storage tier. The user I/O feature library can be built based on these features. Files being used or to be used will be migrated to a higher storage tier in advance or on time.

In general, for hierarchical hybrid storage systems like ONFS, the 'read'/'write' bandwidth of HDD-based storage tier is an order of magnitude lower than DS-d and DS-m, and a majority of data-intensive applications follow features such that they read one entire input file at one time. When user programs are reading or writing the files, the benefit of migrating the files upward in an implicit and automated manner is low, and mostly non-profitable. Therefore, it is better to actively control the upward file migration in an explicit way. For applications that have similar procedures to gene sequencing in Section 7.1(2), we can migrate files automatically. Therefore, it is required to choose the way of migration, either automatic or AUPM, based on real situations.

If 'write' requests come to the source DS during upward migration, then new 'write' data will be stored in the log file. When migration completes, the source DS suspends the access to this file and updates the file in the destination DS with the log file. Then MDS modifies the metadata, and future file accesses will be directed to the destination DS to guarantee data consistency after file migration. Due

to space limitation, we do not provide here detailed analysis and explanation.

7.3 Data consistency during migration

When a file is migrating to a new DSi, any modification to the file will cause inconsistency between the new and old copies. To address this issue, we propose a log-based migration strategy. We take migrating a file from DS-d to DS-m as an example to demonstrate our strategy.

When ONFS starts to migrate a file in DS-d, it first locks the whole file region and copies the file to DS-m through the network. During copying, all 'read'/'write' requests for this file are still served in DS-d. 'Write' requests are recorded in a log file in DS-d since the file has been locked. 'Read' requests first look up the requested data in the log file. If their requested data has been modified, they retrieve the modified data from the log file; otherwise, they are served as normal.

After the file is copied to DS-m, DS-d needs to update all modifications recorded in the log file to the new file in DS-m. Before that, DS-d invalidates all the locks in Clients that have acquired the lock of the file before, and lets Clients flush all dirty data. Meanwhile, DS-d sets a 'migration' flag in the Clients to block all subsequent requests for the file. After the modifications in the log file are updated to the new file in DS-m, DS-d instructs MDS to modify the file metadata to the new data location (i.e., DS-m). The 'migration' flag in Clients is cleared and all blocked requests are resumed. At this time, the migration finishes and all subsequent I/O requests for the file can retrieve the updated metadata in MDS and access the migrated file in DS-m.

8 Evaluations

Lustre is widely used in the HPC environment. ONFS is also designed primarily for HPC applications. We tested the performance of both systems and compare the results. We used I/O benchmark interleaved or random (IOR), MDtest, and some applications to analyze and evaluate the performance of the methods proposed in this study. Experiments were conducted on the TH-1A supercomputer (Yang *et al.*, 2011). The architecture of the TH-1A is shown in Fig. 14.

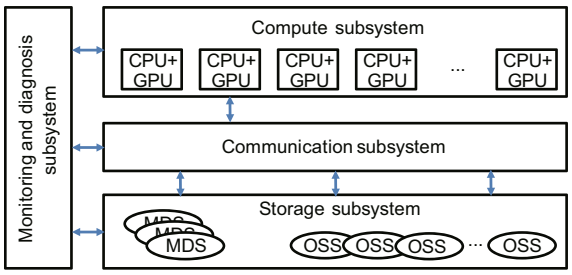

Fig. 14 Architecture of TH-1A
CPU: central processing unit; GPU: graphics processing unit; MDS: metadata server; OSS: object storage server

TH-1A has 7168 compute nodes, each of which is configured with two Intel X5670 CPUs at 2.93 GHz, one NVIDIA GPU, and 48 GB memory (DDR3, frequency 1333 MHz). Some compute nodes have configured SSDs. The 1.2 TB SSD is Intel PR840 with NVMe. TH-1A employs two sets of Lustre, each of which has two PB storage space and one MDS. In Lustre, each object storage server (OSS) has two Intel E5-2620 CPUs and two ARECA1226P RAID cards. Each RAID card manages eight HDDs constituting one object storage target (OST) in RAID6. The HDD is 2 TB with 7200 RPM, 128 MB cache, and an SATA3 interface. The performance of one OST is better than that of one DS-d since DS-d has only HDD and no RAID. Therefore, we used OST to replace DS-d to compare the performance with DS-m in the following experiments. The communication subsystem is self-designed with bi-directional parallel communication. The unidirectional peak bandwidth is 80 Gb/s and it supports RDMA and socket. The Linux kernel version is 3.16.48 and the libfuse version is 2.9.3.

To analyze the impact of low-performance DS-d on user programs' I/O performance during upward and downward migrations, we introduced a storage server, which has two Intel E5-2630 CPU, 64 GB memory, and TH-1A NIC, with 24 independent HDDs to constitute a DS-d (referred to as DS-d24). The HDD in DS-d24 is 4 TB Seagate ST4000NM.

We employed 'collectl' to gather I/O traces from the Client in Lustre, and a software module in ONFS to collect the same I/O traces. In the following tests, we chose a certain number of compute nodes to build DS-m and DS-s. To minimize the impacts of other factors on experimental results, we used direct_IO to disable the page cache in IOR tests. We used DS-m 'read' as an example. The reason is that SSD

has cache, and it is not sensitive to the main parameters of 'write' operations. The DS-m 'read' is fast. Hence, main parameters of 'read' operations can cause obvious impacts on 'read' bandwidth.

8.1 Performance comparison of data servers in three storage tiers

We used the IOR benchmark to test the performance of DS-m, DS-s, and DS-d. We use the performance of OST to replace that of DS-d for comparison.

8.1.1 Performance comparison of a single data server

The performance comparison of a single DS for read and write in each storage tier is shown in Fig. 15.

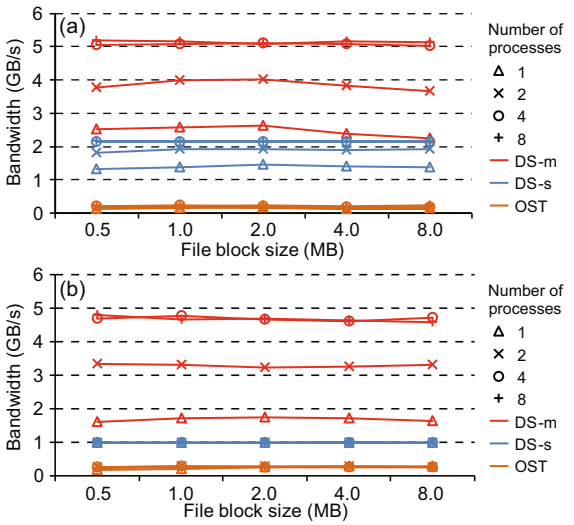

Fig. 15 Comparison of 'read' (a) and 'write' (b) performance of a single data server in each storage tier (OST: object storage target) (References to color refer to the online version of this figure)

The test results show that: (1) the 'read' and 'write' performance of DSs in each storage tier differs a lot. The maximal bandwidths of DS-m and DS-s are about nine times and four times larger than that of OST, respectively; (2) In DS-m, the maximal 'read' bandwidth is about 5000 MB/s under multiple processes. It is restricted by the maximal bandwidth of NIC, which is about 5600 MB/s.

We used synchronous I/O in above tests, where one process sends only one sequential 'read' or 'write' request. Hence, DS-m and DS-s process the 'read'/'write' requests sequentially under a single process. Obviously, the process does not fully explore the high performance potential of memory and SSD through internal parallelism. DS-d (OST) groups eight HDDs to build a RAID, and processes the 'read'/'write' requests in sequence. The bandwidth of RAID is not high. Hence, the difference in bandwidth under different numbers of processes is not obvious. Since DS-m does not have cache, the impact of the number of processes is huge on 'read'/'write' bandwidth. When the number of processes increases from one to four, the 'read'/'write' bandwidth increases linearly. Since SSD has cache, the data to be written are cached. Hence, the number of processes has little effect on the 'write' bandwidth. Though SSD has the pre-fetch mechanism, the aggregated bandwidth of prefetching is limited by the number of prefetch requests that are executed in parallel. Therefore, when the number of processes increases from one to four, the 'read' bandwidth increases accordingly.

8.1.2 Performance comparison of four DSs

To improve the performance of file service with multiple processes and increase the available storage space in the DS-m tier, we propose the method of grouping parallel access control. We test the 'read' and 'write' performance with four OSTs, four DS-s's, and four DS-m's.

The results shown in Fig. 16 reveal that: (1) the amount of differences of aggregated 'read'/'write' bandwidth under multiple processes is more than three times of that between DS-m and DS-s, and more than eight times of that between DS-s and DS-d; (2) With grouping parallel access, the maximal aggregated 'read' bandwidth of four DS-m's reaches around 15 GB/s, which is a huge progress for the storage system; (3) In DS-m, since ONFS is implemented at the user level, the aggregated 'read'/'write' bandwidths with different numbers of processes have many differences. This also shows that there is plenty of room for further improvements in the ONFS prototype system.

8.2 Determining the initial values of *C*n-m and *C*f-m

In Section 6.4, we have discussed the method of determining the initial values of *C*n-m and *C*f-m. The main factors related to determining the initial values of thresholds include parallel 'read'

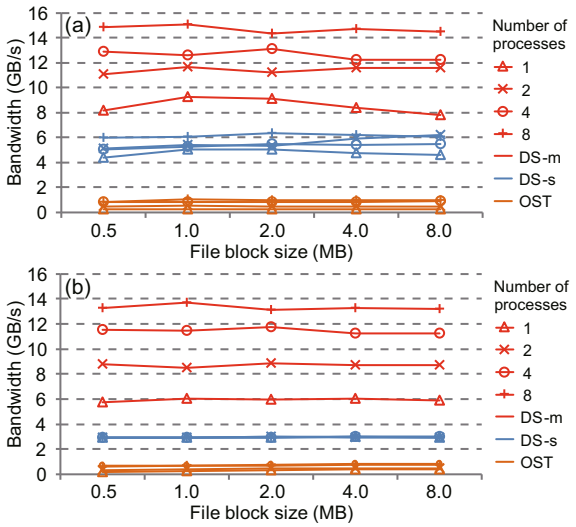

Fig. 16 Comparison of 'read' (a) and 'write' (b) performance of four data servers in each storage tier (References to color refer to the online version of this figure)

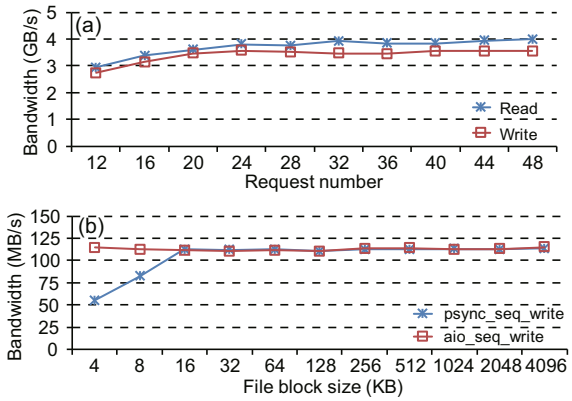

Fig. 17 The aggregated bandwidth of DS-m under multiple processes (a) and the aggregated 'write' bandwidth of DS-d24 under a single process (b)

psync_seq_write: synchronous sequence write; aio_seq_write: asynchronous sequence write

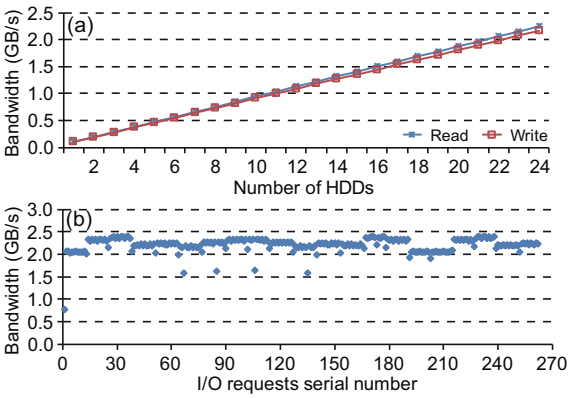

Fig. 18 The aggregated bandwidth of 'write' requests (a) and the changing of 'write' bandwidth over time (b) for 24 HDDs

HDD: hard disk drive; I/O: input/output

(migrating out) and 'write' bandwidth of DS-m, 'write' bandwidth of DS-d24, and the average file size. In the following experiments and analysis, the file size is 8 MB (the reason has been explained in Section 8.6). Fig. 17a shows the aggregated bandwidth of DS-m when executing 'read' and 'write' operations in parallel. On DS-m, the aggregate bandwidths of multiple 'read' and 'write' requests in parallel are approximately 3700 MB/s and 3400 MB/s, respectively. Fig. 17b shows the 'write' bandwidth of DS-d24 under a single process (related to one HDD). When the size of 'write' request is greater than 16 KB, the 'write' bandwidth of a single process is about 112 MB/s.

Fig. 18a presents the aggregated 'write' bandwidth of 24 HDDs on DS-d24 in parallel. The aggregate bandwidth increases linearly with the increasing number of HDDs, and the maximum 'write' bandwidth is 2200 MB/s. For DS-d24, the main factors that affect the aggregated 'write' bandwidth are the performance and the number of HDDs.

Fig. 18b presents the changing of 'write' bandwidth for 24 HDDs in DS-d24. The negative deviation value of 'write' bandwidth is about 2000 MB/s, and the negative deviation is about 9.1%, compared with the average value of 2200 MB/s. It may affect the migration speed of files from DS-m.

To balance the writing speed of DS-m, we migrated out files based on the 'write' bandwidth of DS-m, which is 3400 MB/s. The maximal number of files to be migrated is about 31 (≈3400/112). The total size of the 31 files is 248 MB (31×8 MB), which can be used as the initial value of ΔCn. In the same manner, the initial value of ΔCf can be determined as 24 MB. Since the aggregate 'write' bandwidth of 24 HDDs in DS-d24 has certain negative deviations, the values of Cn-m and Cf-m thresholds should be adjusted dynamically according to practical use.

8.3 Performance of distributed metadata server clusters

8.3.1 Scalability of metadata server cluster

We used MDtest to test the scalability of MDS clusters in ONFS. The results shown in Fig. 19 indicate that the performance of distributed metadata

server clusters in ONFS improves linearly with the increase of MDS.

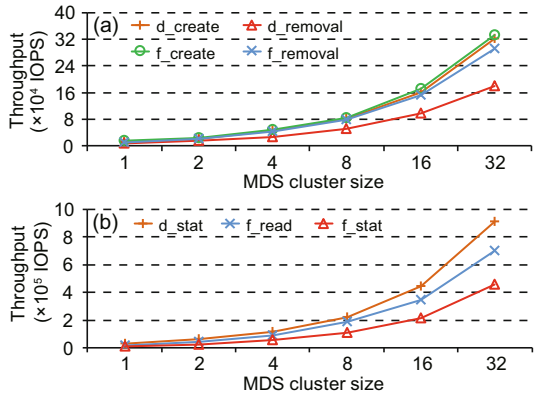

Fig. 19 Scalability of distributed metadata server clusters in on-line and near-line file system (ONFS): (a) metadata modifications; (b) metadata lookups

IOPS: input/output operations per second; d_create: directory create; d_removal: directory removal; f_create: file create; f_removal: file removal; d_stat: directory status; f_read: file read; f_stat: file status

8.3.2 Dynamic adjustment of metadata server loads

There are two main situations in MDS dynamic workload adjustment. One is when the workloads of a single MDS increase temporarily, then a temporary adjustment is needed. The other is when the system-wide metadata workloads increase, then additional MDS is needed.

MDSps shares part of the workloads on MDS-h, and the increased service capacity is determined by the number of UGSDs to be taken out of MDS-h. In Fig. 20, MDS-h and MDSps are equally divided; thus, the total service throughput in MDS-h and MDSps is close to twice that of MDS.

8.4 Overhead of filesystem in userspace

FUSE provides interfaces to user-level file systems and employs techniques such as write-back cache, 128 KB MAX_size, multi-thread, and splicing, to improve the system performance. FUSE divides users' 'read'/'write' requests into sub-requests with a size equal to or smaller than MAX_size. Then the sub-requests are sent to the Client and processed synchronously in a sequential order. The default value of MAX_size is 4 KB, which is originally set for local HDD and too small for faster storage devices. The latencies of processing requests and transmitting data and MAX_size can be the main factors

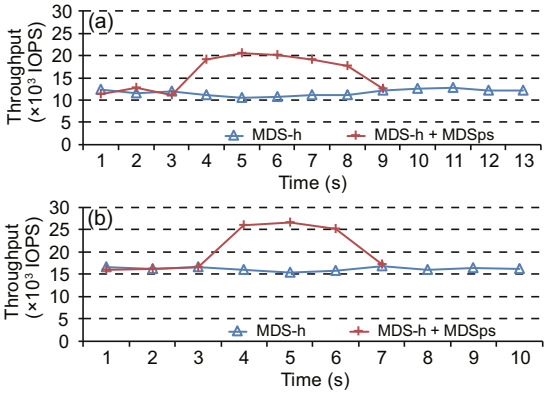

Fig. 20 Performance of workload adjustment of peak-shaving metadata server (MDSps) for 'file create' (a) and 'file read' (b) (IOPS: input/output operations per second)

that impact the performance of FUSE-based file systems.

Vangoor *et al.* (2017) compared the 'read'/'write' performance of FUSE-based Ext4 (StackfsOpt) and native Ext4. The test results show that when the size of I/O is 128 KB or larger, the differences of 'read'/'write' bandwidth between StackfsOpt and native Ext4 are around −2.6%–+2.2%. One exception is that with 128 KB I/O size and rnd-rd-1th-1f, the performance deviation is 12.4%. It indicates that the overhead introduced by FUSE is relatively small.

To analyze the impacts of MAX_size on 'read'/'write' bandwidth, we adjusted the value of MAX_size, and compared the performance with 1 MB and 128 KB MAX_size. MAX_size can be modified to 1 MB by executing the following steps:

```
define FUSE_MAX_PAGES_PER_REQ 256
define MIN_BUFSIZE 0x101000
//After modification, we re-compile the FUSE
    module, and modify the kernel part as follows:
define VM_MAX_READAHEAD 1024
//Then we re-compile the kernel.
```

We conducted experiments to test the 'read' bandwidth of DS-m with 128 KB and 1 MB MAX_size, respectively. The test results are shown in Fig. 21. The average bandwidth with MAX_size of 1 MB is 11.8% higher than that with MAX_size of 128 KB . We also evaluated the performance of DS-s, and the tests show similar results. Therefore, we set MAX_size as 1 MB, which is also aligned to the transfer size between Client and DS-m.

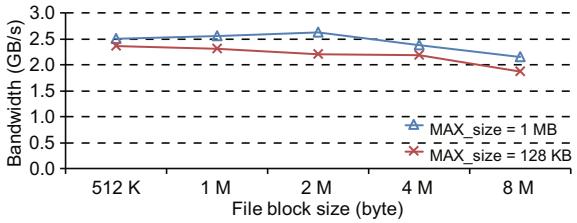

Fig. 21 Bandwidth of sequential DS-m 'read' with MAX_size's of 128 KB and 1 MB

8.5 Impacts of various units on 'read' and 'write' bandwidth during the executions of 'read' and 'write' requests

We take DS-m 'read' as an example to analyze the influences of various units on the 'read' bandwidth during the execution of 'read' requests to guide the optimization and implementation of ONFS. Fig. 22 illustrates the main steps.

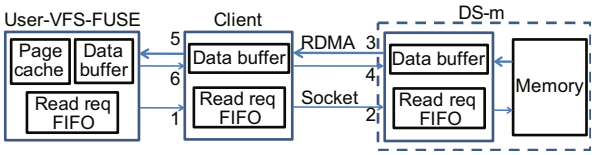

Fig. 22 Main steps during the execution of a 'read' request

FUSE: filesystem in userspace; FIFO: first in, first out; RDMA: remote direct memory access; VFS: virtual file system

In this figure, FIFO stores 'read' requests, and a data buffer stores the requested data. The number in the figure represents the serial number of the time point in one process, referred to as *Ti*. The operations for some time points are described as follows: (1) *T*1: FUSE instructs the Client to obtain a 'read' request; (2) *T*5: the Client instructs FUSE to fetch the data; (3) *T*6: FUSE informs the Client to finish current sub-request. The time period between *Ti* and *Tj* is the time when the unit's finish the particular processes, which is referred to as step *Sij* (*i* represents the beginning time and *j* the finish time). The unit represents hardware components or software modules, such as DS-m and the FUSE module.

As can be seen from the tests, FUSE synchronously processes each sub-request in sequence. To analyze the time spent in each unit during the execution of 'read'/'write' sub-requests and the ratio of it to total time, we use DS-m 'read' as an example to explain that. The results are listed in Table 7.

Table 7 Main steps during the execution of a 'read' request

Step	Operation	Time (μs)	Percentage of time (%)
*S*15	Client receives the 'read' request, until Client and DS-m finish reading	133.3	35.06
*S*56	FUSE obtains the data from Client, then informs Client to finish the request	176.2	46.34
*S*61	FUSE informs Client to finish current 'read' request, until it informs Client to receive a new 'read' request	70.7	18.60

FUSE: filesystem in userspace

The test results reveal that the Client and DS-m take only 35.06% of the total time. FUSE takes 64.94% of total time, which is a relatively large proportion, to obtain data from the Client until instructing the Client to obtain the next 'read' sub-request. The main reason is that FUSE divides a 'read' request with a large size into several sub-requests with a size no larger than MAX_size, and all sub-requests are processed sequentially. In an extreme case that the time spent by the Client and DS-m in step *S*15 is zero, the bandwidth of DS-m 'read' is determined by only FUSE. The Client employs the prefetching mechanism to dramatically decrease the time of reading data from DS-m. If FUSE can process multiple sub-requests in parallel, it is able to decrease the impact of FUSE on the bandwidth of DS-m 'read'.

We also conducted experiments of DS-m 'read' with 1 MB MAX_size on FUSE with the old version, which does not support splice. The time spent on *S*56 in Table 7 is about 243 μs, which is 1.38-fold of the time in FUSE with splice. The time for *S*61 does not change much. Hence, splice is important for reducing the overhead of transmitting data among FUSE, VFS, and users.

8.6 Average file size and the size of storage allocation unit

In ONFS, DS allocates storage space to the 'write' requests. The size of SAU is related to the utilization of storage space, and to the overhead when DS processes a 'write' request. If SAU is too small, then the allocation of an SAU may be too frequent. The average file size has important impacts on determining SAU size.

Wang *et al.* (2004) analyzed the distribution of file sizes on 32 storage servers in an ASCI system. The results reveal that the majority of file sizes are around 2–8 MB. We statistically analyze the number of files according to the distribution of file size in Lustre 'Vol6' in TH-1A. Fig. 23 demonstrates the results.

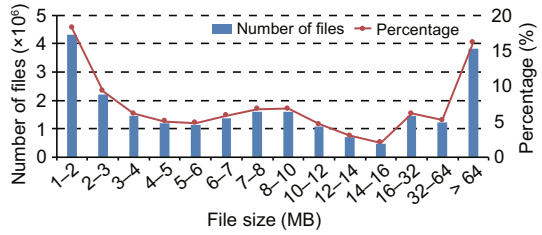

Fig. 23 File size distribution our analysis

By digging into the files with a size less than 1 MB, we find that most are source code, objective files, log files, etc. These files take 80% of the total number of files in Lustre 'Vol6'. One of the main reasons for such a large number is that the majority of files are 'zombie' files that users left in the file system for years. From our understanding of the process of data accessed by users, we can see that most data are asset data. For reasons such as confidentiality, most users will delete the data initiative after using them. In addition, by constraining storage space, the system administrator may force users to manually migrate large files to slower and larger storage devices. Hence, in Lustre 'Vol6', files larger than 1 MB are far fewer than those smaller than 1 MB.

In Fig. 23, there are two peak regions in file size distribution: between 1 MB and 3 MB, and larger than 64 MB. Files between 5 MB and 12 MB are relatively concentrated. We chose 8 MB in this region as the average file size.

Then we discuss how to choose the size of SAU (SAU_size). In Lustre, the default SAU_size is 1 MB (Lofstead *et al.*, 2016), and users can increase it as needed. If SAU_size is too small, e.g., 1 MB, the storage utilization is high, but the overhead of allocation and management may be large. It may easily generate fragmentations in the storage space and makes sequential 'read'/'write' requests random requests, and decreases 'read'/'write' bandwidth. Hence, we propose to select SAU_size as 4 MB. It can be adjusted according to actual situations. To facilitate the migration among three storage tiers, we have employed the same SAU_size in all three storage tiers.

8.7 Benefits of upward migration

We tested and analyzed the benefits of active upward pre-migration. We assume that data is stored in DS-d24 before migration and migrated to DS-m. Fig. 24 illustrates the 'read' bandwidth of DS-m and DS-d in accordance with the time for user programs to process file data. trf*i*-DSd24 is the time when the program reads the *i*th file from DS-d24. trpf*i*-DSd24 is the sum of time in which a file is read from DS-d24 and processed by one process. BW-DSd24 and BW-DSm are the average 'read' bandwidths of DS-d24 and DS-m under one process, respectively. If the time required to process a file is more than the time to read a file from DS-d24, we migrated the next file to be processed to DS-m.

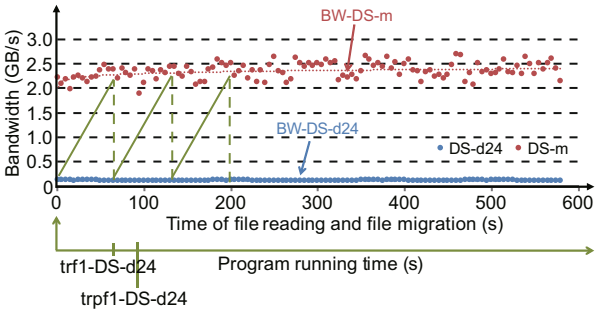

Fig. 24 Bandwidth changes when a file is migrated from DS-d24 to DS-m

The results reveal that BW-DSd24 is around 112 MB/s and BW-DSm can reach 2318 MB/s. With upward migration, the benefit that user programs can obtain for the 'read' requests is about 16 times, which is significant.

Taking the Gathering program (Section 8.8.1) as an example, the total size of one set of input files is around 6720 MB (= 8 × 840 MB) in one processing step, and the time for processing one set of files (=trpf1-DSd24−trf1-DSd24) is about 29 s. The time for migrating one set of input files from DS-d24 to DS-m is about 64 s. Since the time for migrating one set of files is larger than that of processing those files, we begin to pre-migrate from the third set of input files.

During upward migration, there are four main steps: (1) Apply and prepare for data migration.

DS-d sends the upward migration request to MDSS, and MDSS reports to correlated MDS. MDS then form the migration control table, and inform related DS-m and DS-d. (2) DS-d reads files to be migrated from HDD to local buffer. (3) DS-d migrates data from local buffer to DS-m. (4) DS-d informs MDS on the completion of migration, and then MDS modifies the metadata and finishes the migration. Considering that data-intensive programs usually process large files, we ran the experiments with 8 MB files and 128 MB files to represent two situations. Table 8 illustrates the time spent for the first three main steps during upward migration.

Table 8 Time spent for each step in upward migration

File size (MB)	Step	Time (μs)	Percentage of time (%)	Equivalent bandwidth (MB/s)
8	1	126	0.17	
	2	69 258	91.55	116
	3	6263	8.28	1277
128	1	129	0.01	
	2	1 155 198	92.74	111
	3	90 299	7.25	1418

From the test results we can see that the time for DS-d24 to read data from HDD takes 92% of the total time. Therefore, the speed of upward migration is determined mainly by the 'read' bandwidth of HDD, which is about 14 times slower than the 'write' bandwidth of DS-m. In this case, it is important to migrate files that users need to DS-m ahead of time.

Therefore, it is better to migrate files to be accessed by users to DS-m as early as possible. This is also the reason why we proposed upward pre-migration.

8.8 Benefits for typical data-intensive applications

8.8.1 Gathering

We used the Gathering program in an RTM application to verify the benefits of upward migration. RTM is one of the main methods to process petroleum seismic exploration data, and the Gathering program processes result files from RTM. It is a program for production and a typical data-intensive program. For this experiment, it processes 210 000 input files, each of which is around 840 MB. Hence, the total size of the input files is about 176.4 TB. It applies for 512 compute nodes. The whole processing can be divided into many steps, and eight compute nodes are grouped to process 8×8 files together in one step. First, each node reads eight files. Then each node processes the data and generates intermediate results. After that, these eight nodes process the intermediate results together and send the final results to the master node. All input files are numbered and the order of files to be processed by one node is described in a table.

In this experiment, we measured the I/O performance of Lustre (four OSTs with stripe) and ONFS (four DS-m's with grouping) under multiple processes and heavy workloads. We also compared and analyzed the benefits of I/O time and the program's completion time by migrating files from OST in Lustre to DS-m in ONFS. To analyze and compare the performance of ONFS and Lustre, input files that will be processed by one group of nodes are stored in four OSTs (one stripe) and four DS-m's (one Group) for these two systems. Fig. 25 shows the performance of running the Gathering program on OSTs and DS-m's. The I/O traces are collected on one compute node and the results show 10 steps.

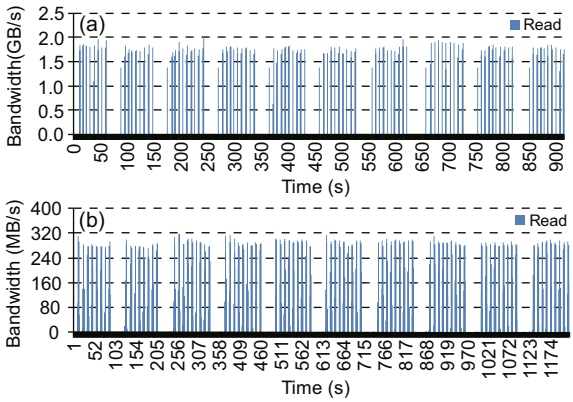

Fig. 25 Input/output performance on the on-line and near-line file system (a) and Lustre (b)

There are two conclusions we can draw from the experiment: (1) The 'read' performance of ONFS is far better than that of Lustre. In one computing step, 64 files are read and the total size is around 53 GB. The 'read' bandwidths of one node to ONFS and Lustre are about 1810 MB/s and 285 MB/s, respectively. The 'read' bandwidth of ONFS is 6.35 times that of Lustre, similar to the IOR results with the same parameters. (2) We can obtain many

benefits from the upward migration. When we migrate files to be processed from OST to DS-m, the 'read' bandwidth achieves a 6.35-fold speedup. In this test case, the I/O time decreases by 303 s, which takes 24.9% of the total time when running the program on Lustre. The acceleration effect on program operation is obvious. In fact, it is related to the ratio of I/O operation time to computation time. This shows that for data-intensive applications, it is important to substantially improve I/O performance.

8.8.2 One-way wave depth migration

This example shows a program for oil seismic data processing: one-way wave depth migration (OWDM), which is also a program for production. The number of input files to be processed is 30 170 and the average size of files is about 80 MB. The number of final results (local image files) is 30 170 and the average size of files is about 940 MB. It applies for N compute nodes to run the program, and each node processes $30\,170/N$ input files. The main steps of processing an input file are shown in Table 9.

Table 9 Main steps of processing an input file in orthogonal wavelength division multiplexing

Step	Operation	Number of files	File size (MB)	Total size (MB)
$S1$	Read an input file	1	80	80
$S2$	Read a parameter file	1	60	60
$S3$	Write a parameter file	1	850	850
$S4$	Write intermediate results	514	254	130 556
$S5$	Write a local image file	1	940	940
$S6$	Read total image1 file	1	3000	3000
$S7$	Write total image2 file	1	2000	2000

In the above steps, the total amounts of 'read' and 'write' data are about 3.14 GB and 134 GB, respectively. Since I/O and computing are independent among nodes, we used 64 compute nodes for processing and testing. All files to be processed by one node are stored in one stripe (four OSTs) in Lustre and one Group (four DS-m's) in ONFS.

In $S4$, the intermediate results are the computing results from a frequency domain, and are stored for the program to resume execution when it is in error. It is similar to the checkpointing data. Regardless of the physical characteristics of the program and the programming method, OWDM is a typical data-intensive and write-intensive program. Figs. 26 and 27 show the results of OWDM performance running on Luster and ONFS, respectively.

The 'write' bandwidth in the middle range of the total time when running OWDM is summarized in Table 10. From the results, ONFS 'write' bandwidth is 4.67 times large as Luster. Under the cache on/off situation, the total times of running OWDM in ONFS are only 86% and 24% of the time when running in Lustre, respectively. Obviously, the performance of ONFS is much better than that of Luster. Especially with cache off, all I/O requests from OWDM program flooded into the storage system, which brings a huge I/O pressure to the storage system.

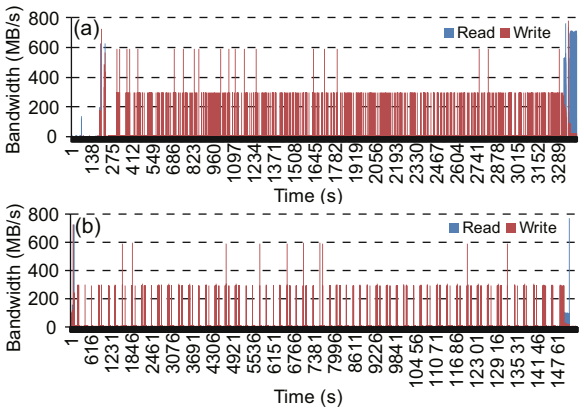

Fig. 26 Performance of the orthogonal wavelength division multiplexing on Lustre with cache on (a) and cache off (b) (References to color refer to the online version of this figure)

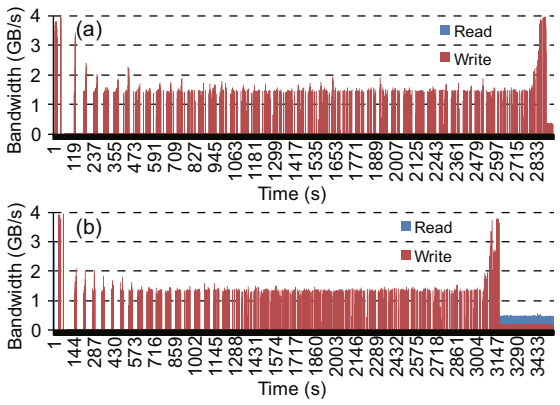

Fig. 27 Performance of orthogonal wavelength division multiplexing on the on-line and near-line file system with cache on (a) and cache off (b) (References to color refer to the online version of this figure)

In Lustre, because of low 'read' and 'write' bandwidth, the I/O time of the program increases

Table 10 Summary about OWDM applications

File system	Cache on/off	'Write' bandwidth (MB/s)	Run-time (s)	Percentage of run-time (%)
ONFS	On	1500	2927	86
	Off	1400	3552	24
Lustre	On	300	3416	
	Off	300	15 111	

ONFS: on-line and near-line file system; OWDM: orthogonal wavelength division multiplexing

significantly. Hence, the time for running the program in Lustre is 4.25 times longer than that in ONFS. This demonstrates that the performance of the storage system has a very important impact on the performance of data-intensive applications. In this experiment, we set cache off for creating a test environment closer to a practical application. If the program occupies most memory of nodes, the size of the page cache will reduce, and its effects will reduce greatly, or to an extreme case that the cache does not work anymore. The Gathering program is a read-intensive and 'read'/'write' mixed test case, and the OWDM program is a write-intensive test case. From the results of these two experiments, we know that ONFS has a very good balanced 'read' and 'write' performance.

8.9 'Read'/'write' performance tests on large-scale systems

We run an IOR benchmark to test the scalability of each storage tier that consists of 32 data servers. The block size is 1 MB, and the number of processes varies from 32 to 256. The average size of data accessed from each data server is around 2 GB. The results shown in Fig. 28 indicate that both DS-m and DS-s have the scalability as good as Lustre and the aggregated bandwidth of DS-m and DS-s is far better than that of OST.

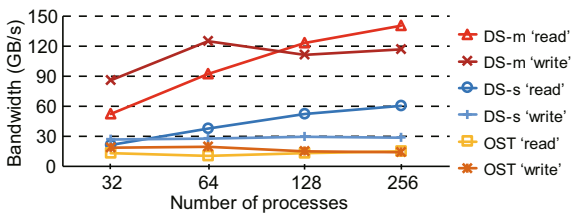

Fig. 28 Scalability of each storage tier (OST: object storage target)

This shows that the performance of memory-based and SSD-based storage systems has great advantages over an HDD-based storage system. The 'read' performance of all three types of data servers is improving with the increase of the number of processes. However, the 'write' performance reaches the best with 64 processes, where each data server is accessed by two processes. If the number of processes continues to grow, the 'write' performance will setback. This is because more processes occupy the cache or buffer in data servers. This degrades a cache's acceleration effects on programs.

8.10 Comparison with state-of-the-art storage systems

To validate the performance benefits of ONFS, it is necessary to compare it with state-of-the-art storage systems based on fast storage media. We selected four recent related studies or products to compare with ONFS, including FusionFS (Zhao *et al.*, 2014), Cray DataWarp (Ovsyannikov *et al.*, 2017), Gordon (Strande *et al.*, 2012), and Cray Sonexion 3000 (Cray, 2017). Not FusionFS, but Cray DataWarp and Cray Sonexion 3000 have already been deployed in Cori in the National Energy Research Scientific Computing Center (NERSC), and Gordon is a deployed system in the San Diego Supercomputer Center (SDSC). Thus, we used their information on real systems in Table 11. For FusionFS, we used the information of the experiments on Intrepid in Zhao *et al.* (2014).

FusionFS is a distributed file system based on memory in compute nodes. It saves an extreme amount of data movement between compute and storage resources by storing data in memory. The chief objection to FusionFS is that it uses memory in compute nodes without considering performance impacts on jobs in the nodes.

Cray DataWarp installs SSDs on the burst buffer nodes and organizes them with a temporal file system. It handles peaks and spikes in I/O bandwidth requirements and reduces job wall clock time. Its main shortcoming compared with ONFS is that user jobs must use job script directives or APIs to manually stage in or out files.

Gordon is a data-intensive supercomputer with 1024 compute nodes. Each group of 16 compute nodes is connected with an I/O node. SSDs are located in I/O nodes, acting as burst buffers. The

Table 11 Comparison of ONFS with the state-of-the-art storage systems

Storage system	Storage-media	Location	System features	Bandwidth (single node or OST) (GB/s)	Current system size	Scalability	Weakpoints (compared with ONFS)
ONFS	DRAM +SSD	Compute node	Hierarchical storage system, automatic migration	5.10	$O(N)$[3]	Scale to all compute nodes	NULL
FusionFS	DRAM	Compute node	Memory-based storage system	0.16[1]	$O(N)$[3]	Scale to all compute nodes	May impact job performance
DataWarp (CoriBBN)	SSD	Burst buffer node	Fast temporal storage system	5.90[2]	288 BBNs	Scale to limited burst buffer nodes	Manual migration
Gorden	SSD	I/O node	Localfile system in ION	2.60	64 IONs	Scale to limited I/O nodes	Limited scalability
Sonexion 3000 (Cori Lustre)	SSD +HDD	Storage-node	SSD as a cache of OST	1.40	248 OSTs	Scale to limited OSTs	Limited scalability

ONFS: on-line and near-line file system; DRAM: dynamic random access memory; SSD: solid state drive; HDD: hard disk drive; OST: object storage target

[1] The performance of FusionFS is tested with HDD in the paper. If tested with DRAM, it can perform much better with a speed larger than 0.16 GB/s

[2] The performance of DataWarp is the parallel bandwidth of four SSDs in the burst buffer node

[3] N represents the number of compute nodes in TH-1A and Intrepid

main shortcoming is its limited scale of deployment, since SSDs can be placed in only a limited number of I/O nodes.

Cray latest (HDD+SSD)-based Lustre Sonexion 3000 puts two SSDs in each of its SSU, acting as the cache of eight OSTs. Although it achieves better performance than HDD-based Lustre, its main limitation is that SSD can be deployed in only hundreds of OSTs, leading to limited aggregated bandwidth. In contrast, ONFS leverages idle memory in the system, and can scale to tens of thousands of compute nodes.

To summarize, ONFS provides persistent storage space to applications with high bandwidth and low latency by transparently migrating files among memory-based, SSD-based, and HDD-based storage tiers. It dynamically borrows idle memory from compute nodes without impacting the jobs running on them. ONFS has better parallel performance and scalability since it can easily scale to all the compute nodes in the system, which is much more than I/O nodes and storage servers.

9 Conclusions

The US Department of Energy and seven leading US national laboratories initiated a project named FFSIO to work on solutions to currently intractable I/O problems of extreme-scale systems. FFSIO suggests absorbing a tremendous amount of I/O traffic in the data staging areas closer to applications by integrating SSD in the compute nodes or I/O nodes. Although some solutions have already been deployed in current supercomputers (e.g., DataWarp in Cori), how to effectively leverage this fast storage tier has not been well-studied yet, and remains an open topic.

As an effort to promote the use of the SSD storage tier, we have developed a hierarchical hybrid file system called 'ONFS' to manage the node-local SSDs and HDD-based storage servers. Furthermore, we proposed to borrow the underused memory commonly existing in compute nodes to construct a faster memory-based storage tier that is located closer to applications. ONFS manages the three storage tiers in a unified namespace and migrates files among them dynamically and transparently. We proposed several techniques to enhance the performance of

ONFS and conducted extensive tests on TH-1A with multiple micro-benchmarks and real applications to validate our ideas. Results show that ONFS has significant speedups.

Currently, ONFS is implemented with FUSE. In the future, we plan to re-implement it in the kernel to further minimize the software overhead. In addition, we plan to study in depth the caching and prefetching strategies of the client-side cache in ONFS.

References

Agrawal, N., Bolosky, W.J., Douceur, J.R., *et al.*, 2007. A five-year study of file-system metadata. *ACM Trans. Stor.*, **3**(3):9. https://doi.org/10.1145/1288783.1288788

ALCF, 2017. Computational Systems: Mira. Argonne Leadership Computing Facility. https://www.alcf.anl.gov/user-guides/computational-systems

Ali, N., Carns, P., Iskra, K., *et al.*, 2009. Scalable I/O forwarding framework for high-performance computing systems. IEEE Int. Conf. on CLUSTER Computing and Workshops, p.1-10.

Anderson, E., Hall, J., Hartline, J., *et al.*, 2001. An experimental study of data migration algorithms. Proc. Algorithm Engineering, Int. Workshop, p.145-158.

Appuswamy, R., van Moolenbroek, D.C., Tanenbaum, A.S., 2012. Integrating flash-based SSDs into the storage stack. IEEE Symp. on Mass Storage Systems and Technologies, p.1-12.

Bent, J., Grider, G., Kettering, B., *et al.*, 2012. Storage challenges at Los Alamos National Lab. IEEE 28th Symp. on Mass Storage Systems and Technologies, p.1-5. https://doi.org/10.1109/MSST.2012.6232376

Bharathi, S., Chervenak, A., Deelman, E., *et al.*, 2008. Characterization of scientific workflows. 3rd Workshop on Workflows in Support of Large-Scale Science, p.1-10. https://doi.org/10.1109/WORKS.2008.4723958

Byan, S., Lentini, J., Madan, A., *et al.*, 2012. Mercury: host-side flash caching for the data center. IEEE 28th Symp. on MASS Storage Systems and Technologies, p.1-12. https://doi.org/0.1109/MSST.2012.6232368

Canim, M., Mihaila, G.A., Bhattacharjee, B., *et al.*, 2010. SSD bufferpool extensions for database systems. *Proc. VLDB Endow.*, **3**(1-2):1435-1446. https://doi.org/10.14778/1920841.1921017

Carns, P.H., Ligon, W.B., III, Ross, R.B., 2000. PVFS: a parallel file system for Linux clusters. Proc. 4th Annual Linux Showcase and Conf., p.317-328.

Carns, P.H., Harms, K., Allcock, W., *et al.*, 2011. Understanding and improving computational science storage access through continuous characterization. *ACM Trans. Stor.*, **7**(3):1-14. https://doi.org/10.1145/2027066.2027068

Chen, F., Koufaty, D.A., Zhang, X., 2011. Hystor: making the best use of solid state drives in high performance storage systems. Proc. Int. Conf. on Supercomputing, p.22-32. https://doi.org/10.1145/1995896.1995902

Cheong, S.K., Jeong, J.J., Jeong, Y.W., *et al.*, 2011. Research on the I/O performance advancement of a low speed HDD using DDR-SSD. 6th Int. Conf. on Future Information Technology, p.508-513. https://doi.org/10.1007/978-3-642-22333-4_66

Congiu, G., Narasimhamurthy, S., Süß, T., *et al.*, 2016. Improving collective I/O performance using non-volatile memory devices. IEEE Int. Conf. on Cluster Computing, p.120-129. https://doi.org/10.1109/CLUSTER.2016.37

Cray, 2017. Cray Sonexion 3000. https://www.cray.com/products/storage/sonexion

Dai, N., Wu, W., Zhang, W., *et al.*, 2011. TTI RTM using variable grid in depth. Int. Petroleum Technology Conf., p.1-7. https://doi.org/10.2523/IPTC-15050-MS

Dell EMC, 2017. All Flash Storage. https://www.dellemc.com/en-us/storage/discover-flash-storage/index.htm

Dong, W.R., Liu, G.M., Yu, J., *et al.*, 2015. SFDC: file access pattern aware cache framework for high-performance computer. IEEE 17th Int. Conf. on High Performance Computing and Communications, IEEE 7th Int. Symp. on Cyberspace Safety and Security, IEEE 12th Int. Conf. on Embedded Software and Systems, p.342-350. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.185

Dong, X., Xie, Y., Muralimanohar, N., *et al.*, 2011. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, **8**(2):1-29. https://doi.org/10.1145/1970386.1970387

Dongarra, J., 2010. Impact of architecture and technology for extreme scale on software and algorithm design. Department of Energy Workshop on Cross-cutting Technologies for Computing at the Exascale.

Facebook, 2013. Flashcache at Facebook from 2010 to 2013 and Beyond. https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920/

Gluster, 2017. Gluster File System. http://www.gluster.org

Hitachi Data Systems Cooperation, 2010. Dynamic Storage Tiering: the Integration of Block, File and Content. https://shobiziems.com/hitachi_nas/hitachi-white-paper-dynamic-storage-tiering.pdf

Holland, D.A., Angelino, E., Wald, G., *et al.*, 2013. Flash caching on the storage client. USENIX Annual Technical Conf., p.127-138.

IBM, 2017. IBM Blue Gene/Q https://www-03.ibm.com/systems/technicalcomputing/solutions/bluegene/

Intel, 2017. Intel Data Center SSD. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-SSDs/dc-p4600-series/dc-p4600-4tb-aic-3d1.html

Iskra, K., Romein, J.W., Yoshii, K., *et al.*, 2008. ZOID: I/O-forwarding infrastructure for petascale architectures. Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, p.153-162. https://doi.org/10.1145/1345206.1345230

Kim, Y., Gupta, A., Urgaonkar, B., *et al.*, 2011. Hybridstore: a cost-efficient, high-performance storage system combining SSDs and HDDs. IEEE 19th Annual Int. Symp. on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, p.227-236. https://doi.org/10.1109/MASCOTS.2011.64

Kuhlen, M., Vogelsberger, M., Angulo, R., 2012. Numerical simulations of the dark universe: state of the art and the next decade. *Phys. Dark Univ.*, **1**(1):50-93. https://doi.org/10.1016/j.dark.2012.10.002

Lee, D., Choi, J., Kim, J.H., *et al.*, 1999. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. Proc. ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems, p.134-143. https://doi.org/10.1145/301453.301487

Liao, X., Xiao, L., Yang, C., *et al.*, 2014. MilkyWay-2 supercomputer: system and application. *Front. Comput. Sci.*, **8**(3):345-356. https://doi.org/10.1007/s11704-014-3501-3

Liu, N., Cope, J., Carns, P., *et al.*, 2012. On the role of burst buffers in leadership-class storage systems. IEEE 28th Symp. on MASS Storage Systems and Technologies, p.1-11. https://doi.org/10.1109/MSST.2012.6232369

Liu, X., Lu, Y., Yu, J., *et al.*, 2017a. MemUsing: dynamic, efficient memory utilization in compute nodes for HPC memory-based storage systems. Proc. 7th Int. Workshop on Computer Science and Engineering, p.8-16.

Liu, X., Lu, Y., Wu, C., *et al.*, 2017b. UGSD: scalable and efficient metadata management for EB-scale file systems. Proc. Int. Conf. on Compute and Data Analysis, p.81-90. https://doi.org/10.1145/3093241.3093257

LLNL, 2012. Sequoia. Lawrence Livermore National Laboratory. https://computation.llnl.gov/computers/sequoia

Lofstead, J., Jimenez, I., Maltzahn, C., *et al.*, 2016. DAOS and friends: a proposal for an exascale storage system. Int. Conf. for High Performance Computing, Networking, Storage & Analysis, p.585-596. https://doi.org/10.1109/SC.2016.49

Lu, C.Y., Alvarez, G.A., Wilkes, J., 2002. Aqueduct: online data migration with performance guarantees. FAST Conf. on File and Storage Technologies, p.219-230.

Miller, E.L., Greenan, K., Leung, A., *et al.*, 2011. Reliable and efficient metadata storage and indexing using nvram. *J. Comput. Sci. Technol.*, **26**(3):344-351.

Muralidhar, S., Lloyd, W., Roy, S., *et al.*, 2014. f4: Facebook's warm blob storage system. Proc. 11th USENIX Symp. on Operating Systems Design and Implementation, p.383-398.

NERSC, 2017a. Burst Buffer Architecture and Software Roadmap. National Energy Research Scientific Computing Center. http://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer

NERSC, 2017b. The Configuration of Cori File System. National Energy Research Scientific Computing Center. http://www.nersc.gov/users/computational-systems/cori/configuration/

NetApp, 2016. All Flash Arrays. http://www.netapp.com/us/products/storage-systems/all-flash-array/aff-a-series.aspx

Ocaña, K., de Oliveira, D., 2015. Parallel computing in genomic research: advances and applications. *Adv. Appl. Bioinform. Chem.*, **8**:23-25. https://doi.org/10.2147/AABC.S64482

Ovsyannikov, A., Romanus, M., Straalen, B.V., *et al.*, 2017. Scientific workflows at datawarp-speed: accelerated data-intensive science using NERSE's burst buffer. Joint Int. Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, p.1-6. https://doi.org/10.1109/PDSW-DISCS.2016.005

Pawlowski, B., Juszczak, C., Staubach, P., *et al.*, 1994. NFS version 3: design and implementation. USENIX Summer Technical Conf., p.137-152.

Prabhakar, R., Vazhkudai, S.S., Kim, Y., *et al.*, 2011. Provisioning a multi-tiered data staging area for extreme-scale machines. Int. Conf. on Distributed Computing Systems, p.1-12. https://doi.org/10.1109/ICDCS.2011.33

Qiao, F., Song, Z., Bao, Y., *et al.*, 2013. Development and evaluation of an earth system model with surface gravity waves. *J. Geophys. Res. Ocean.*, **118**(9):4514-4524. https://doi.org/10.1002/jgrc.20327

Rajachandrasekar, R., Moody, A., Mohror, K., *et al.*, 2013. A 1 PB/s file system to checkpoint three million MPI tasks. Proc. 22nd Int. Symp. on High-Performance Parallel and Distributed Computing, p.143-154. https://doi.org/10.1145/2462902.2462908

Rodeh, O., Teperman, A., 2003. zFS—a scalable distributed file system using object disks. Proc. 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, p.207-218. https://doi.org/10.1109/MASS.2003.1194858

Roselli, D., Anderson, T.E., Lorchid, J.R., 2000. A comparison of file system workloads. Proc. USENIX Annual Technical Conf., p.41-54.

Saito, S., Oikawa, S., 2012. Exploration of non-volatile memory management in the OS kernel. 3rd Int. Conf. on Networking and Computing, p.302-306. https://doi.org/10.1109/ICNC.2012.56

Sato, K., Mohror, K., Moody, A., *et al.*, 2014. A user-level infiniband-based file system and checkpoint strategy for burst buffers. 14th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, p.21-30. https://doi.org/10.1109/CCGrid.2014.24

Satyanarayanan, M., Kistler, J.J., Kumar, P., *et al.*, 1990. Coda: a highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, **39**(4):447-459. https://doi.org/10.1109/12.54838

Saxena, M., Swift, M.M., Zhang, Y., 2012. FlashTier: a lightweight, consistent and durable storage cache. Proc. 7th ACM European Conf. on Computer Systems, p.267-280. https://doi.org/10.1145/2168836.2168863

Schenck, W., El Sayed, S., Foszczynski, M., *et al.*, 2017. Evaluation and performance modeling of a burst buffer solution. *ACM SIGOPS Oper. Syst. Rev.*, **50**(1):12-26. https://doi.org/10.1145/3041710.3041714

Schmuck, F., Haskin, R., 2002. GPFS: a shared-disk file system for large computing clusters. Proc. 1st USENIX Conf. on File and Storage Technologies, No. 19.

Seagate Technology LLC, 2017. Seagate NAS+SRS HDD Product Manual. https://www.seagate.com/www-content/product-content/nas-fam/nas-hdd/en-us/docs/100764115g.pdf

Shalf, J., Dosanjh, S., Morrison, J., 2010. Exascale computing technology challenges. Int. Conf. on High Performance Computing for Computational Science, p.1-25.

Shibata, T., Choi, S., Taura, K., 2010. File-access patterns of data-intensive workflow applications and their implications to distributed filesystems. Proc. 19th ACM Int. Symp. on High Performance Distributed Computing, p.746-755. https://doi.org/10.1145/1851476.1851585

Soundararajan, G., Prabhakaran, V., Balakrishnan, M., *et al.*, 2010. Extending SSD lifetimes with disk-based write caches. Proc. 8th USENIX Conf. on File and Storage Technologies, No. 8.
https://doi.org/10.1021/ja0386501

Strande, S.M., Cicotti, P., Sinkovits, R.S., *et al.*, 2012. Gordon: design, performance, and experiences deploying and supporting a data intensive supercomputer. Proc. 1st Conf. of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond, No. 3.
https://doi.org/10.1145/2335755.2335789

Tan, Z., Zhou, W., Feng, D., *et al.*, 2013. ALDM: adaptive loading data migration in distributed file systems. *IEEE Trans. Magn.*, **49**(6):2645-2652.
https://doi.org/10.1109/TMAG.2013.2251616

Uta, A., Sandu, A., Kielmann, T., 2016. Overcoming data locality. *Fut. Gener. Comput. Syst.*, **54**(C):144-158.
https://doi.org/10.1016/j.future.2015.01.013

Vangoor, B.K.R., Tarasov, V., Zadok, E., 2017. To FUSE or not to FUSE: performance of user-space file systems. Proc. 15th USENIX Conf. on File and Storage Technologies, p.59-72.

Vetter, J.S., Mittal, S., 2015. Opportunities for nonvolatile memory systems in extreme-scale high-performance computing. *Comput. Sci. Eng.*, **17**(2):73-82.
https://doi.org/10.1109/MCSE.2015.4

Wang, F., Xin, Q., Hong, B., *et al.*, 2004. File system workload analysis for large scale scientific computing applications. Proc. 21st IEEE/12th NASA Goddard Conf. on Mass Storage Systems and Technologies, p.139-152.

Wang, F., Oral, S., Shipman, G., *et al.*, 2010. Understanding Lustre Filesystem Internals. Technical Report, No. ORNL/TM-2009/117. Oak Ridge National Laboratory, National Center for Computational Sciences, Oak Ridge, USA.

Wang, T., Oral, S., Wang, Y., *et al.*, 2014. BurstMem: a high-performance burst buffer system for scientific applications. IEEE Int. Conf. on Big Data, p.71-79.
https://doi.org/10.1109/BigData.2014.7004215

Wang, T., Mohror, K., Moody, A., *et al.*, 2016. An ephemeral burst-buffer file system for scientific applications. Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, p.807-818.
https://doi.org/10.1109/SC.2016.68

Weil, S.A., Brandt, S.A., Miller, E.L., *et al.*, 2006. Ceph: a scalable, high-performance distributed file system. Proc. 7th Symp. on Operating Systems Design and Implementation, p.307-320.

Yang, X.J., Liao, X.K., Lu, K., *et al.*, 2011. The TianHe-1A supercomputer: its hardware and software. *J. Comput. Sci. Technol.*, **26**(3):344-351.
https://doi.org/10.1007/s02011-011-1137-8

Yildiz, O., Dorier, M., Ibrahim, S., *et al.*, 2016. On the root causes of cross-application I/O interference in HPC storage systems. IEEE Int. Parallel and Distributed Processing Symp., p.750-759.
https://doi.org/10.1109/IPDPS.2016.50

Yu, J., Liu, G.M., Dong, W.R., *et al.*, 2017. WatCache: a workload-aware temporary cache on the compute side of HPC systems. *J. Supercomput.*, **1**(2):1-33.
https://doi.org/10.1007/s11227-017-2167-7

Zhao, D.F., Raicu, I., 2013. HyCache: a user-level caching middleware for distributed file systems. IEEE Int. Symp. on Parallel and Distributed Processing Workshops and Phd Forum, p.1997-2006.
https://doi.org/10.1109/IPDPSW.2013.83

Zhao, D.F., Zhang, Z., Zhou, X.B., *et al.*, 2014. FusionFS: toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. IEEE Int. Conf. on Big Data, p.61-70.
https://doi.org/10.1109/BigData.2014.7004214

Xin LIU, first author of this invited paper, received her Bachelor's degree in Computer Science and Technology from the Nanjing University of Science and Technology in China in 2010. She is a PhD candidate in Computer Science and Technology at the National University of Defense Technology in China, and a PhD candidate at the University of Nebraska-Lincoln in the USA. She studies high-performance storage and parallel file system, and participates in many projects supported by Chinese funding agencies.

Dr. Yu-tong LU, ISC fellow, a faculty at both Sun Yat-sen University (SYSU) and the National University of Defense Technology (NUDT). She is Director of National Supercomputing Center in Guangzhou. She received her B.S., MS and PhD from the NUDT. She has extensive research and development experience over several generations of Chinese supercomputers, and she is the deputy chief designer of Tianhe-2 system. She leads a number of HPC and big data projects under the support of the Chinese Ministry of Science and Technology, the National Natural Science Foundation of China, and Guangdong Province. Her research interests include parallel operating system (OS), high speed communications, global file systems, and advanced programming environments converging HPC and big data.